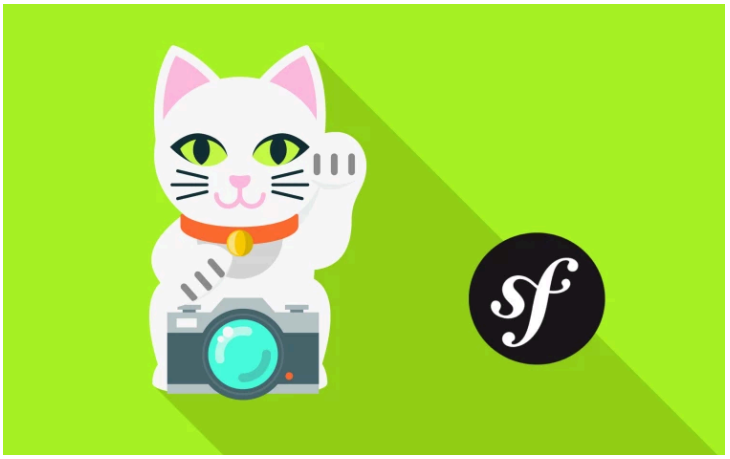
# Messenger! Queue work for Later



# Chapter 1: Installing Messenger

Yo Friends! It's Symfony Messenger time!!! So, what *is* Symfony Messenger? It's a tool that allows you to... um... send messages... Wait... that made no sense.

## Um, What *is* Messenger?

Let's try again. Messenger is a tool that enables a *really* cool design pattern where you write "messages" and then other code that *does* something when that message is sent. If you've heard of CQRS - Command Query Responsibility Segregation - Messenger is a tool that enables that design pattern.

That's all great... and we're going to learn *plenty* about it. But there's a good chance you're watching this because you want to learn about something *else* that Messenger does: it allows you to run code asynchronously with queues & workers! OooooOOoo. That's the *real* fanciness of Messenger.

Oh, and I have two more sales pitches. First, Symfony 4.3 has a *ton* of new features that *really* make Messenger shine. And second, using Messenger is an absolute delight. So... let's do this!

## Project Setup

If you want to become a command-bus-queue-processing-worker-middleware-envelope... and other buzzwords... Messenger *master*, warm up your coffee and code along with me. Download the course code from this page. When you unzip it, you'll find a `start/` directory inside with the same code that you see here. Open up the `README.md` file for *all* the details about how to get the project running *and* a totally-unrelated, yet, lovely poem called "The Messenger".

The last setup step will be to find a terminal and use the Symfony binary to start a web-server at `https://localhost:8000`:

```
symfony serve
```

Ok, let's go check that out in our browser. Say hello to our newest SymfonyCasts creation: Ponka-fy Me. If you didn't already know, Ponka, by day, is one of the lead developers here at SymfonyCasts. By night... she is Victor's cat. Actually... due to her frequent nap schedule... she doesn't really *do* any coding... now that I think about it.

## Ponka-fy Me

Anyways, we've been noticing a problem where we go on vacation, but Ponka can't come... so when we return, none of our photos have Ponka in them! Ponka-fy Me solves that: let's select a vacation photo... it uploads... and... yea! Check it out! Ponka *seamlessly* joined us in our vacation photo!

Behind the scenes, this app uses a Vue.js frontend... which isn't important for what we'll be learning. What *is* important to know is that this uploads to an API endpoint which stores the photo and then *combines* two images together. That's a pretty heavy thing to do on a web request... which is why, if you watch closely, it's kinda slow: it will finish uploading... wait... and, yep, *then* load the new image on the right.

Let's look at the API endpoint so you can get an idea of how this works: it lives at `src/Controller/ImagePostController.php`. Look for `create()` *this* is the upload API endpoint: it grabs the file, validates it, uses *another* service to store that file - that's the `uploadImage()` method, creates a new `ImagePost` entity, saves it to the database with Doctrine and *then*, down here, we have some code to add Ponka to our photo. That `ponkafy()` method does the *really* heavy-lifting: it takes the two images, splices them together and... to make it extra dramatic and slow-looking for the purposes of this tutorial, it takes a 2 second break for tea.

Mostly... all of this code is meant to be *pretty* boring. Sure, I've organized things into a few services... that's nice - but it's all very traditional. It's a *perfect* test case for Messenger!

## Installing Messenger

So... let's get it installed! Find your terminal, open a new tab and run:

```
composer require messenger
```

When that finishes... we get a "message"... from Messenger! Well, from its recipe. This is great - but we'll talk about all this stuff along the way.

In addition to installing the Messenger component, its Flex recipe made two changes to our app. First, it modified `.env`. Let's see... it added this "transport" config. This relates to queuing messages - a lot more on that later.

.env

```
// ... lines 1 - 29
###> symfony/messenger ###
# Choose one of the transports below
# MESSENGER_TRANSPORT_DSN=amqp://guest:guest@localhost:5672/%2f/messages
# MESSENGER_TRANSPORT_DSN=doctrine://default
# MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
###
```

It also added a new `messenger.yaml` file, which... if you open that up... is *perfectly*... boring! It has `transports` and `routing` keys - again, things that relate to queuing - but it's all empty and doesn't do *anything* yet.

config/packages/messenger.yaml

```
framework:
    messenger:
        # Uncomment this (and the failed transport below) to send failed messages to this transport for later handling.
        # failure_transport: failed

        transports:
            # https://symfony.com/doc/current/messenger.html#transports
            # async: '%env(MESSENGER_TRANSPORT_DSN)%'
            # failed: 'doctrine://default?queue_name=failed'
            # sync: 'sync://'

        routing:
            # Route your messages to the transports
            # 'App\Message\YourMessage': async
```

So... what *did* installing the Messenger component give us... other than some new PHP classes inside the `vendor/` directory? It gave us one new important service. Back at your terminal run:

```
php bin/console debug:autowiring mess
```

There it is! We have a new service that we can use with this `MessageBusInterface` type-hint. Um... what does it do? I don't know! But let's find out next! Along with learning about message classes and message handlers.

# Chapter 2: Message, Handler & the Bus

Messenger is what's known as a "Message Bus"... which is kind of a generic tool that can be used to do a couple of different, but similar design patterns. For example... Messenger can be used as a "Command bus", a "Query bus", an "Event bus" *or*... a "School bus". Oh... wait... that last one was never implemented... ok, it can be used for the first three. Anyways, if these terms mean absolutely *nothing* to you... great! We'll talk about what all of this means along the way.

## Command Bus Pattern

Most people will use Messenger as a "command bus"... which is sort of a design pattern. Here's the idea. Right now, we're doing all of our work in the controller. Well, ok, we've organized things into services, but our controller calls those methods directly. It's nicely-organized, but it's still basically procedural: you can read the code from top to bottom.

With a command bus, you separate *what* you want to happen - called a "command" - from the code that *does* that work. Imagine you're working as a waiter or waitress at a restaurant and someone wants a pizza margherita... with extra fresh basil! Mmm. Do you... run back to the kitchen and cook it yourself? *Probably* not... Instead, you write down the order. But... let's say instead, you write down a "command": cook a pizza, margherita style with extra fresh basil. Next, you "send" that command to the kitchen. And finally, some chef does all the magic to get that pizza ready. Meanwhile, you're able to take more orders and send more "commands" back to the kitchen.

*This* is a command bus: you create a simple, informational command "cook a pizza", give it to some central "system"... which is given that fancy word "bus", and *it* makes sure that something sees that command and "handles" it... in this case, a "chef" cooks the pizza. And that central "bus" is probably smart enough to have different people "handle" different commands: the chef cooks the pizza, but the bar tender prepares the drink orders.

## Creating the Command Class

Let's recreate that *same* idea... in code! The "command" *we* want to issue is: add Ponka to this image. In Messenger, each command is a simple PHP class. In the `src/` directory, create a new `Message/` directory. We can put our command, or "message", classes anywhere... but this is a nice way to organize things. Create a new PHP class called `AddPonkaToImage`... because that describes the *intent* of what we want to happen: we want someone to add ponka to the image. Inside... for now... do *nothing*.

src/Message/AddPonkaToImage.php

```
↕ // ... lines 1 - 2
3 namespace App\Message;
4
5 class AddPonkaToImage
6 {
7 }
```

A message class is *your* code: it can look *however* you want. More on that later.

## Creating the Handler Class

Command, done! Step 2 is to create the "handler" class - the code that will *actually* add Ponka to an image. Once again, this class can live anywhere, but let's create a new `MessageHandler/` directory to keep things organized. The handler class can *also* be called anything... but unless you *love* being confused... call it `AddPonkaToImageHandler`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 2
3 namespace App\MessageHandler;
↕ // ... lines 4 - 5
6 use Symfony\Component\Messenger\Handler\MessageHandlerInterface;
7
8 class AddPonkaToImageHandler implements MessageHandlerInterface
9 {
↕ // ... lines 10 - 13
14 }
```

Unlike the message, the handler class *does* have a few rules. First, a handler class must implement `MessageHandlerInterface`... which is actually *empty*. It's a "marker" interface. We'll talk about *why* this is needed in a bit. And second, the class must have a public function called `__invoke()` with a single argument that is *type-hinted* with the message class. So,

`AddPonkaToImage`, then any argument name: `$addPonkaToImage`. Inside, hmm, just to see how this all works, let's `dump($addPonkaToImage)`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 4
5  use App\Message\AddPonkaToImage;
↕ // ... lines 6 - 7
8  class AddPonkaToImageHandler implements MessageHandlerInterface
9  {
10     public function __invoke(AddPonkaToImage $addPonkaToImage)
11     {
12         dump($addPonkaToImage);
13     }
14 }
```

## Connecting the Message and Handler

Ok, let's back up. On a high level, here's how this is going to work. In our code, we'll create an `AddPonkaToImage` object and tell messenger - the message bus - to "handle" it. Messenger will see our `AddPonkaToImage` object, go get the `AddPonkaToImageHandler` service, call its `__invoke()` method and pass it the `AddPonkaToImage` object. That's... all there is to it!

But wait... how does messenger know that the `AddPonkaToImage` object should be "handled" by `AddPonkaToImageHandler`? Like, if we had multiple command and handler classes, how would it know which handler handles which message?

Find your terminal and run:

```
php bin/console debug:messenger
```

This is an *awesome* command: it shows us a map of which handler will be called for each message. We only have 1 right now, but... yea, somehow it *already* knows that `AddPonkaToImage` should be handled by `AddPonkaToImageHandler`. How?

It knows thanks to two things. First, that empty `MessageHandlerInterface` is a "flag" that tells Symfony that this is a messenger "handler". And second, Messenger looks for a method called `__invoke()` and reads the *type-hint* on its argument to know *which* message class this should handle. So, `AddPonkaToImage`.

And yes, you can *totally* configure all of this in a different way, and even skip adding the interface by using a tag. We'll talk about some of this later... but it's usually not something you need to worry about.

Oh, and if you're not familiar with the `__invoke()` method, ignoring Messenger for a minute, that's a magic method you can put on any PHP class to make it "executable": you can take an object and call it like a function... *if* it has this method:

```
$handler = new AddPonkaToImageHandler();
$handler($addPonkaToImage);
```

That detail is not important *at all* to understand Messenger, but it explains why this, otherwise "strange" method name was chosen.

## Dispatching the Message

Phew! Status check: we have a message class, we have a handler class, and thanks to some smartness from Symfony, Messenger knows these are linked together. The *last* thing we need to do is... actually send the command, or "message", to the bus!

Head over to `ImagePostController`. This is the endpoint that uploads our image and adds Ponka to it. Fetch the message bus by adding a new argument with the `MessageBusInterface` type-hint.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 15
16 use Symfony\Component\Messenger\MessageBusInterface;
↕ // ... lines 17 - 21
22 class ImagePostController extends AbstractController
23 {
↕ // ... lines 24 - 38
39     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, PhotoPonkaficator $ponkaficator, MessageBusInterface $messageBus)
40     {
↕ // ... lines 41 - 77
78     }
↕ // ... lines 79 - 109
110 }
```

Then... right *above* all the Ponka image code - we'll leave all of that there for the moment - say `$message = new AddPonkaToImage()`. And then `$messageBus->dispatch($message)`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 5
6 use App\Message\AddPonkaToImage;
↕ // ... lines 7 - 21
22 class ImagePostController extends AbstractController
23 {
↕ // ... lines 24 - 38
39     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, PhotoPonkaficator $ponkaficator, MessageBusInterface $messageBus)
40     {
↕ // ... lines 41 - 60
61         $message = new AddPonkaToImage();
62         $messageBus->dispatch($message);
63 
64         /*
65          * Start Ponkafication!
66          */
↕ // ... lines 67 - 77
78     }
↕ // ... lines 79 - 109
110 }
```

That's it! `dispatch()` is the *only* method on that object... it doesn't get any more complicated than this.

So... let's try it! If everything works, this `AddPonkaToImage` object should be passed to `__invoke()` and then we'll dump it. Since this will all happen on an AJAX request, we'll use a trick in the profiler to see if it worked.

Head back and refresh the page... just to be safe. Upload a new photo and... when it finishes, down on the web debug toolbar, hover over the arrow icon to find... nice! Here is that AJAX request. I'll hold Command and click the link to open it in a new tab. This is the profiler for that AJAX request. Click the "Debug" link on the left.

Ha! There it is! This shows us that our `dump()` code *was* executed during the AJAX request! It worked! We pass the message to the message bus and then it calls the handler.

Of course... our handler doesn't *do* anything yet. Next, let's move all of the Ponkafication logic from our controller into the handler.

# Chapter 3: Doing Work in the Handler

Inside our controller, after we save the new file to the filesystem, we're creating a new `AddPonkaToImage` object and dispatching it to the message bus... or technically the "command" bus... because we're currently using it as a command bus. The end result is that the bus calls the `__invoke()` method on our handler and passes it that object. Messenger understands the connection between the message object and handler thanks to the argument type-hint and this interface.

## Command Bus: Beautifully Disappointing

By the way, you might be thinking:

> *“Wait... the whole point of a "command" bus is to... just "call" this `__invoke()` method for me? Couldn't I just... ya know... call it myself and skip a layer?”*

And... yes! It's *that* simple! It *should* feel completely underwhelming at first!

But having that "layer", the "bus", in the middle gives us two nice things. First, out code is more decoupled: the code that creates the "command" - our controller in this case - doesn't know or care about our handler. It dispatches the message and moves on. And second, this *simple* change is going to allow us to execute handlers asynchronously. More on that soon.

## Moving code into the Handler

Back to work: all the code to add Ponka to the image is *still* done inside our controller: this gets an updated version of the image with Ponka inside, another service actually *saves* the new image onto the filesystem, and this last bit - `$imagePost->markAsPonkaAdded()` - updates a date field on the entity. It's only a few lines of code... but that's a lot of work!

Copy all of this, remove it, and I'll take my comments out too. Paste all of that into the handler. Ok, no surprise, we have some undefined variables. `$ponkaficator`, `$photoManager` and `$entityManager` are all services.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 7
8  class AddPonkaToImageHandler implements MessageHandlerInterface
9  {
10     public function __invoke(AddPonkaToImage $addPonkaToImage)
11     {
12         $updatedContents = $ponkaficator->ponkafy(
13             $photoManager->read($imagePost->getFilename())
14         );
15         $photoManager->update($imagePost->getFilename(),
   $updatedContents);
16         $imagePost->markAsPonkaAdded();
17         $entityManager->flush();
18     }
19 }
```

In the controller... on top, we were autowiring those services into the controller method. We don't need `$ponkaficator` anymore.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 20
21 class ImagePostController extends AbstractController
22 {
↕ // ... lines 23 - 37
38     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
39     {
↕ // ... lines 40 - 63
64     }
↕ // ... lines 65 - 95
96 }
```

Anyways, how can we get those services in our handler? Here's the really cool thing: the "message" class - `AddPonkaToImage` is a simple, "model" class. It's kind of like an entity: it doesn't live in the container and we don't autowire it into our classes. If we need an `AddPonkaToImage` object, we say: `new AddPonkaToImage()`. If we decide to give that class any constructor arguments - more on that soon - we pass them right here.

But the *handler* classes are *services*. And *that* means we can use, good, old-fashioned dependency injection to get any services we need.

Add `public function __construct()` with, let's see here, `PhotoPonkaficator $ponkaficator`, `PhotoFileManager $photoManager` and... we

need the entity manager: `EntityManagerInterface $entityManager`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 5
6  use App\Photo\PhotoFileManager;
7  use App\Photo\PhotoPonkaficator;
8  use Doctrine\ORM\EntityManagerInterface;
↕ // ... lines 9 - 10
11 class AddPonkaToImageHandler implements MessageHandlerInterface
12 {
↕ // ... lines 13 - 16
17     public function __construct(PhotoPonkaficator $ponkaficator,
   PhotoFileManager $photoManager, EntityManagerInterface $entityManager)
18     {
↕ // ... lines 19 - 21
22     }
↕ // ... lines 23 - 32
33 }
```

I'll hit `Alt + Enter` and select Initialize Fields to create those properties and set them.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 10
11 class AddPonkaToImageHandler implements MessageHandlerInterface
12 {
13     private $ponkaficator;
14     private $photoManager;
15     private $entityManager;
16
17     public function __construct(PhotoPonkaficator $ponkaficator,
   PhotoFileManager $photoManager, EntityManagerInterface $entityManager)
18     {
19         $this->ponkaficator = $ponkaficator;
20         $this->photoManager = $photoManager;
21         $this->entityManager = $entityManager;
22     }
↕ // ... lines 23 - 32
33 }
```

Now... let's use them: `$this->ponkaficator`, `$this->photoManager`, `$this->photoManager` again... and `$this->entityManager`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 10
11 class AddPonkaToImageHandler implements MessageHandlerInterface
12 {
↕ // ... lines 13 - 23
24     public function __invoke(AddPonkaToImage $addPonkaToImage)
25     {
26         $updatedContents = $this->ponkaficator->ponkafy(
27             $this->photoManager->read($imagePost->getFilename())
28         );
29         $this->photoManager->update($imagePost->getFilename(),
   $updatedContents);
↕ // ... line 30
31         $this->entityManager->flush();
32     }
33 }
```

## Message Class Data

Nice! This leaves us with just one undefined variable: the actual `$imagePost` that we need to add Ponka to. Let's see... in the controller, we create this `ImagePost` entity object... which is pretty simple: it holds the filename on the filesystem... and a few other minor pieces of info. This is what we store in the database.

Back in `AddPonkaToImageHandler`, at a high level, this class needs to know *which* `ImagePost` it's supposed to be working on. How can we *pass* that information from the controller to the handler? By putting it on the message class! Remember, this is *our* class, and it can hold *whatever* data we want.

So now that we've discovered that our handler needs the `ImagePost` object, add a `public function __construct()` with one argument: `ImagePost $imagePost`. I'll do my usual Alt+Enter and select "Initialize fields" to create and set that property.

src/Message/AddPonkaToImage.php

```
↕ // ... lines 1 - 4
5  use App\Entity\ImagePost;
6
7  class AddPonkaToImage
8  {
9      private $imagePost;
10
11     public function __construct(ImagePost $imagePost)
12     {
13         $this->imagePost = $imagePost;
14     }
↕ // ... lines 15 - 19
20 }
```

Down below, we'll need a way to *read* that property. Add a getter: `public function getImagePost()` with an `ImagePost` return type. Inside, `return $this->imagePost`.

src/Message/AddPonkaToImage.php

```
↕ // ... lines 1 - 6
7  class AddPonkaToImage
8  {
↕ // ... lines 9 - 15
16     public function getImagePost(): ImagePost
17     {
18         return $this->imagePost;
19     }
20 }
```

And really... you can make this class look however you want: we could have made this a `public` property with no need for a constructor or getter. Or you could replace the constructor with a `setImagePost()`. *This* is the way I like to do it... but it doesn't matter: as long as it holds the data you want to pass to the handler... you're good!

Anyways, now we're dangerous! Back in `ImagePostController`, down here, `AddPonkaToImage` *now* needs an argument. Pass it `$imagePost`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 20
21 class ImagePostController extends AbstractController
22 {
↕ // ... lines 23 - 37
38     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, MessageBusInterface $messageBus)
39     {
↕ // ... lines 40 - 59
60         $message = new AddPonkaToImage($imagePost);
↕ // ... lines 61 - 63
64     }
↕ // ... lines 65 - 95
96 }
```

Then, over in the handler, finish this with `$imagePost = $addPonkaToImage->getImagePost()`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 10
11 class AddPonkaToImageHandler implements MessageHandlerInterface
12 {
↕ // ... lines 13 - 23
24     public function __invoke(AddPonkaToImage $addPonkaToImage)
25     {
26         $imagePost = $addPonkaToImage->getImagePost();
↕ // ... lines 27 - 32
33     }
34 }
```

I love it! So that's the *power* of the message class: it really *is* like you're writing a message to someone that says:

> *"I want you to do a task and here's all the information that you need to know to do that task."*

Then, you hand that off to the message bus, it calls the handler, and the handler has all the info it needs to do that work. It's a simple... but really neat idea.

Let's make sure it all works: move over and refresh just to be safe. Upload a new image and... it still works!

Next: there's already one other job we can move to a command-handler system: *deleting* an image.

# Chapter 4: Message, Handler & debug:messenger

Our app has one other small superpower. If for *some* reason you're not happy with your Ponka image... I'm not even sure *how* that would be possible... you can delete it. When you click that button, it sends an AJAX request that hits this `delete()` action.

And... that really does two things. First, `$photoManager->deleteImage()` takes care of physically deleting the image from the filesystem. I added a `sleep()` for dramatic effect, but deleting something from the filesystem *could* be a bit heavy if the files were stored in the cloud, like on S3.

And second, the controller deletes the `ImagePost` from the database. But... thinking about these two steps... the only thing we need to do *immediately* is delete the image from the database. If we *only* did that and the user refreshed the page, it *would* be gone. And then... if we deleted the actual file a few seconds... or minutes or even *days* later... that would be totally fine! But... more on doing fancy asynchronous stuff in a few minutes.

## Creating DeleteImagePost

Right now, let's refactor all this deleting logic into the command bus pattern we just learned. First, we need the message, or "command" class. Let's copy `AddPonkaToImage`, paste and call it `DeleteImagePost.php`. Update the class name and then... um... do *nothing*! Coincidentally, this message class will look *exactly* the same: the handler will need to know *which* `ImagePost` to delete.

src/Message/DeleteImagePost.php

```
// ... lines 1 - 2
namespace App\Message;

use App\Entity\ImagePost;

class DeleteImagePost
{
    private $imagePost;

    public function __construct(ImagePost $imagePost)
    {
        $this->imagePost = $imagePost;
    }

    public function getImagePost(): ImagePost
    {
        return $this->imagePost;
    }
}
```

## Creating DeleteImagePostHandler

Time for step 2 - the handler! Create a new PHP class and call it `DeleteImagePostHandler`. Like before, give this a `public function __invoke()` with a `DeleteImagePost` type-hint as the only argument.

src/MessageHandler/DeleteImagePostHandler.php

```
// ... lines 1 - 2
namespace App\MessageHandler;

use App\Message\DeleteImagePost;

class DeleteImagePostHandler
{
    public function __invoke(DeleteImagePost $deleteImagePost)
    {
    }
}
```

Now, it's the same process as before: copy the first three lines of the controller, delete them, and paste them into the handler. This time, we need two services.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 6
7  class DeleteImagePostHandler
8  {
9      public function __invoke(DeleteImagePost $deleteImagePost)
10     {
11         $photoManager->deleteImage($imagePost->getFilename());
12
13         $entityManager->remove($imagePost);
14         $entityManager->flush();
15     }
16 }
```

Add `public function __construct()` with `PhotoFileManager $photoManager` and `EntityManagerInterface $entityManager`. I'll hit Alt + Enter and click initialize fields to create both of those properties and set them.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 5
6  use App\Photo\PhotoFileManager;
7  use Doctrine\ORM\EntityManagerInterface;
8
9  class DeleteImagePostHandler
10 {
11     private $photoManager;
12     private $entityManager;
13
14     public function __construct(PhotoFileManager $photoManager,
   EntityManagerInterface $entityManager)
15     {
16         $this->photoManager = $photoManager;
17         $this->entityManager = $entityManager;
18     }
↕ // ... lines 19 - 27
28 }
```

Down here, use `$this->photoManager`, `$this->entityManager` and one more `$this->entityManager`. And, like before, we need to know which `ImagePost` we're deleting. Prep that with `$imagePost = $deleteImagePost->getImagePost()`.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 8
9  class DeleteImagePostHandler
10 {
↕ // ... lines 11 - 19
20     public function __invoke(DeleteImagePost $deleteImagePost)
21     {
22         $imagePost = $deleteImagePost->getImagePost();
23         $this->photoManager->deleteImage($imagePost->getFilename());
24
25         $this->entityManager->remove($imagePost);
26         $this->entityManager->flush();
27     }
28 }
```

## Dispatching the Message

Ding! That's my... it's done sound! Because, we have a message, a handler and Symfony *should* know that they're linked together. The *last* step is to *send* the message. In the controller... we don't need these last two arguments anymore... we *only* need `MessageBusInterface $messageBus`. And then, this is *wonderful*, our *entire* controller is: `$messageBus->dispatch(new DeleteImagePost($imagePost))`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 6
7  use App\Message\DeleteImagePost;
↕ // ... lines 8 - 21
22 class ImagePostController extends AbstractController
23 {
↕ // ... lines 24 - 69
70     public function delete(ImagePost $imagePost, MessageBusInterface
   $messageBus)
71     {
72         $messageBus->dispatch(new DeleteImagePost($imagePost));
73
74         return new Response(null, 204);
75     }
↕ // ... lines 76 - 93
94 }
```

Pretty cool, right? Let's see if it all works. Move over, click the "x" and... hmm... it didn't disappear. And... it looks like it was a 500 error! Through the power of the profiler, we can click the little link to jump *straight* to a big, beautiful, HTML version of that exception. Interesting:

## Command Bus: Each Message should have One Handler

> *"No handler for message* `App\Message\DeleteImagePost`*"*

That's interesting. Before we figure out what went wrong, I want to mention one thing: in a command bus, each message *normally* has exactly *one* handler: not two and not zero. And *that's* why Messenger gives us a helpful error if it can't find that handler. We'll talk more about this later and *bend* these rules when we talk about event buses.

## Debugging the Missing Handler

Anyways... why does Messenger think that `DeleteImagePost` doesn't have a handler? Can't it see the `DeleteImagePostHandler` class? Find your terminal and run:

```
php bin/console debug:messenger
```

Woh! It only sees our *one* handler class! What this command *really* does is this: it finds *all* the "handler" classes in the system, then prints the "message" that it handles next to it. So... this confirms that, for some reason, Messenger doesn't see our handler!

And... you may have spotted my mistake! To find all the handlers, Symfony looks in the `src/` directory for classes that implement `MessageHandlerInterface`. And... I forgot that part! Add `implements MessageHandlerInterface`.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\Messenger\Handler\MessageHandlerInterface;
9
10 class DeleteImagePostHandler implements MessageHandlerInterface
11 {
↕ // ... lines 12 - 28
29 }
```

Run `debug:messenger` again:

```
php bin/console debug:messenger
```

*Now* it sees it! Let's try it again: close up the profiler, try hitting "x" and... this time it works!

Status report: we have two messages and each has a handler that's potentially doing some pretty heavy work, like image manipulation or talking across a network if files are stored in the cloud. It's time to talk about *transports*: the key concept behind taking this work and doing it asynchronously so that our users don't have to wait for all that heavy work to finish before getting a response.

# Chapter 5: Transport: Do Work Later (Async)

So far, we've separated the instructions of what we want to do - we want to add Ponka to this `ImagePost` - from the logic that actually does that work. And... it's a nice coding pattern: it's easy to test and if we need to add Ponka to an image from anywhere else in our system, it will be super pleasant.

But this pattern unlocks some *serious* possibilities. Think about it: now that we've isolate the *instructions* on what we want to do, instead of handling the command object immediately, couldn't we, in theory, "save" that object somewhere... then read and process it later? That's... basically how a queuing system works. The *advantage* is that, depending on your setup, you could put less load on your web server *and* give users a faster experience. Like, right now, when a user clicks to upload a file, it takes a few seconds before it finally pops over here. It's not the biggest deal, but it's not ideal. If we can fix that easily, why not?

## Hello Transports

In Messenger, the key to "saving work for later" is a system called transports. Open up `config/packages/messenger.yaml`. See that `transports` key? The details are actually configured in `.env`.

Here's the idea: we're going to say to Messenger:

> *"Yo! When I create an `AddPonkaToImage` object, instead of handling it immediately, I want you to send it somewhere else."*

That "somewhere else" is a transport. And a transport is usually a "queue". If you're new to queueing, the idea is *refreshingly* simple. A queue is an external system that "holds" onto information in a big list. In our case, it will hold onto serialized message objects. When we send it another message, it adds it to the list. Later, you can read those messages from the queue one-by-one, handle them and, when you're done, the queue will remove it from the list.

Sure... robust queuing systems have a lot of other bells and whistles... but that really *is* the main concept.

## Transport Types

There are a *bunch* of queueing systems available, like RabbitMQ, Amazon SQS, Kafka, and queueing at the supermarket. Out-of-the box, Messenger supports three: `amqp` - which basically means RabbitMQ, but *technically* means any system that implements the "AMQP" spec - `doctrine` and `redis`. AMQP is the most powerful... but unless you're already a queueing pro and want to do something crazy, these all work exactly the same.

Oh, and if you need to talk to some unsupported transport, Messenger integrates with another library called Enqueue, which supports a bunch more.

## Activating the doctrine Transport

Because I'm already using Doctrine in this project, let's use the `doctrine` transport. Uncomment the environment variable for that.

.env

```
↕ // ... lines 1 - 29
30 ###> symfony/messenger ###
↕ // ... lines 31 - 32
33 MESSENGER_TRANSPORT_DSN=doctrine://default
↕ // ... line 34
35 ###
```

See this `://default` part? That tells the Doctrine transport that we want to use the `default` Doctrine connection. Yep, it'll re-use the connection you've already set up in your app to store the message inside a new table. More on that soon.

**Tip**

Starting in symfony 5.1, the code behind the Doctrine transport was moved to its own package. The only difference is that you should now also run this command: `composer require symfony/doctrine-messenger`

Now, back in `messenger.yaml`, uncomment this `async` transport, which *uses* that `MESSENGER_TRANSPORT_DSN` environment variable we just created. The name - `async` - isn't important - that could be anything. But, in a second, we'll start *referencing* that name.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 5
        transports:
            # https://symfony.com/doc/current/messenger.html#transports
            async: '%env(MESSENGER_TRANSPORT_DSN)%'
// ... lines 9 - 16
```

## Routing to Transports

At this point... yay! We've told Messenger that we have an `async` transport. And if we want back and uploaded a file *now*, it would... make absolutely no difference: it would *still* be processed immediately. Why?

Because we need to *tell* Messenger that *this* message should be sent to that transport, instead of being handled right now.

Back in `messenger.yaml`, see this `routing` key? When we dispatch a message, Messenger looks at *all* of the classes in this list... which is zero right now if you don't count the comment... and looks for our class - `AddPonkaToImage`. If it doesn't find the class, it handles the message immediately.

Let's tell Messenger to *instead* send that to the `async` transport. Set `App\Message\AddPonkaToImage` to `async`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 12
        routing:
            # Route your messages to the transports
            'App\Message\AddPonkaToImage': async
```

As *soon* as we do that... it makes a *huge* difference. Watch how *fast* the image loads on the right after uploading. Boom! That was faster than before and... Ponka isn't there! Gasp!

Actually, let's try one more - that first image was a *little* bit slow because Symfony was rebuilding its cache. This one should be nearly instant. It is! Instead of calling our handler immediately, Messenger is *sending* our message to the Doctrine transport.

## Seeing the Queued Message

And... um... what does that actually mean? Find your terminal... or whatever tool you like to use to play with databases. I'll use the `mysql` client to connect to the `messenger_tutorial` database. Inside, let's:

```
SHOW TABLES;
```

Woh! We expected `migration_versions` and `image_post`... but suddenly we have a *third* table called `messenger_messages`. Let's see what's in there:

```
SELECT * FROM messenger_messages;
```

Nice! It has two rows for our *two* messages! Let's use the magic `\G` to format this nicer:

```
SELECT * FROM messenger_messages \G
```

Cool! The `body` holds our object: it's been serialized using PHP's `serialize()` function... though that can be configured. The object is wrapped inside something called an `Envelope`... but inside... we can see our `AddPonkaToImage` object and the `ImagePost` inside of that... complete with the filename, `createdAt` date, etc.

Wait... but where did this table come from? By default, if it's not there, Messenger creates it for you. If you don't want that, there's a config option called `auto_setup` to disable this - I'll show you how later. If you *did* disable auto setup, you could then use the handy `setup-transports` command on deploy to create that table for you.

```
php bin/console messenger:setup-transports
```

This doesn't do anything now... because the table is already there.

Hey! This was a *huge* step! Whenever we upload images... they are *not* being handled immediately: when we upload two more... they're being sent to Doctrine and *it* is keeping track of them. Thanks Doctrine!

Next, it's time to *read* those messages one-by-one and start handling them. We do that with a console command called a "worker".

# Chapter 6: Worker Command

Even if I refresh the page, now that our messages aren't being handled immediately... the four most recent photos don't have Ponka in them. That's tragic! Instead, those messages were sent to the `doctrine` transport and are waiting patiently inside of a `messenger_messages` table.

So... how can we read these back out and process them? We need something that can fetch each row one-by-one, *deserialize* each message back into PHP, then pass it to the message bus to be *actually* handled. That "thing" is a special console command. Run:

```
php bin/console messenger:consume
```

You won't see any output... yet... but, unless we messed something up, this is doing exactly what we need: reading each message, deserializing it, and sending it *back* to the bus for handling.

So... let's go refresh. Woh! It *did* work! All 4 messages now have Ponka on them! We're saved!

## messenger:consume -vv

To make this more interesting, as you can see, it says to run this command with `-vv` if you want to see what it's doing behind-the-scenes. But... interesting, once the command finished reading and handling all 4 messages... it didn't quit: it's *still* running. And if we restart it with `-vv` on the end:

```
php bin/console messenger:consume -vv
```

... it does the same. A command that "handles" messages from a queue is called a "worker". And the job of a worker is to watch and wait for new messages to be added to the queue and handle them the *instant* one is added. It waits and runs... forever! Well, that's not *totally* true - but more on that later when we talk about deployment.

Let's peek back over in our "queue" - the `messenger_messages` table:

```
SELECT * FROM messenger_messages \G
```

Yep! This holds *zero* rows because all those messages were processed and removed from the queue. Back at the browser, let's upload... how about... 5 new photos. Woh... that was *awesome* fast!

Ok, ok, move back to the terminal that's running the worker! We can see it doing its job! It says: "Received message", "Message handled by `AddPonkaToImageHandler`" then "`AddPonkaToImage` was handled successfully (acknowledging)". That last part, "acknowledging" means that Messenger notified the Doctrine transport that the message was handled and can be removed from the queue.

Then... it keeps going to the next message... and the next... and the next... until it's done. So if we refresh... Ponka was added to all of these! Let's do it again - upload 5 more photos. And... let's refresh and watch... there's Ponka! We can see them being handled little-by-little. So much wonderful Ponka!

Ok, this *would* be cooler if our JavaScript automatically refreshed the image when Ponka was added... instead of me needing to refresh the page... but that's a *totally* different topic, and one that's covered by the Mercure component in Symfony.

And... that's it! This `messenger:consume` command is something that you'll have running on production *all* the time. Heck, you might decide to run *multiple* worker processes. *Or*, you could *even* deploy your app to a totally *different* server - one that's not handling web requests - and run the worker processes there! *Then*, handling these messages wouldn't use *any* resources from your web server. We'll talk more about deployment later.

## Problem: Database Didn't Update?

Because right now... we have a problem... a kinda weird problem. Refresh the page. Hmm, the original photos all say something like:

> *“Ponka visited 13 minutes ago. Ponka visited 11 minutes ago.”*

But, since we made things asynchronous, these all say:

> *“Ponka is napping. Check back soon.”*

Open up the `ImagePost` entity and find the `$ponkaAddedAt` property. This is a `datetime` field, which records *when* Ponka was added to the photo. The message on the front-end comes from this value.

For the original ones... back when the whole process was synchronous, this field *was* set successfully. But now... it looks like it isn't. Let's check the database to be sure. Over in MySQL, run:

```
SELECT * FROM image_post \G
```

All the way back in the beginning... `ponka_added_at` *was* being set. But now they're all `null`. So... our images are being processed correctly, but, for some reason, this field in the database is *not*. If we look inside `AddPonkaToImageHandler`... yea... right here: `$imagePost->markPonkaAsAdded()`. *That* sets the property. So... why isn't it saving?

Let's figure out what's going on next and learn a bit more about some "best practices" when it comes to building your message class.

# Chapter 7: Problems with Entities in Messages

We've got a strange issue: we know that `AddPonkaToImageHandler` is being called successfully by the worker process.... because it's *actually* adding Ponka to the images! But, for some reason... even though we call `$imagePost->markAsPonkaAdded()`... which sets the `$ponkaAddedAt` property... and then `$this->entityManager->flush()`... it doesn't seem to be saving!

## Maybe we're Missing persist()?

So.. you might wonder:

> *"Do I need to call persist() on `$imagePost`?"*

Let's try it: `$this->entityManager->persist($imagePost)`. In theory, we should *not* need this: you only need to call `persist()` on *new* objects that you want to save. It's not needed... and normally does *nothing*... when you call it on an object that will be updated.

But... what the heck... let's see what happens.

## Restarting the Worker

Oh! But before we try this... we need to do something *very* important! Find your terminal, press Ctrl+C to stop the worker, then restart it:

```
php bin/console messenger:consume
```

Why? As you know, workers sit there and run... forever. The *problem* is that, if you update some of your code, the worker won't see it! Until you restart it, it still has the *old* code stored in memory! So anytime you make a change to code that a worker uses, be sure to restart it. Later, we'll talk about how to do this safely when you deploy.

## The Weirdness of Serialized Entities

Let's see what happens now that we've added that new `persist()` call. Upload one new file, find your worker and... yep! It was handled successfully. Did that fix the entity saving problem? Refresh the page.

Yikes! What just happened! The image shows up *twice*! One *with* the date set... and one without. To the database!

```
SELECT * FROM image_post \G
```

Yea... this one image is on *two* rows: I know because they're pointing to the *exact* same file on the filesystem. The worker... somehow... *duplicated* that row in the database.

## Doctrine's Identity Map

This... is a confusing bug... but it has an easy fix. First, let's look at things from Doctrine's perspective. Internally, Doctrine keeps track of a list of all the entity objects that it's currently dealing with. When you query for an entity, it adds it to this list. When you call `persist()`, if it's not already in the list, it's added. *Then*, when we call `flush()`, Doctrine loops over *all* of these objects, looks for any that changed, and creates the appropriate UPDATE or INSERT queries. It knows whether or not an object should be inserted or updated because it knows whether or not *it* was responsible for *querying* for that object. By the way, if you want to nerd out on this topic more, this "list" is called the identity map... and it's just a big array that starts empty at the beginning of each request and gets bigger as you query or persist things.

So now let's think about what happens in our worker. When it deserializes the `AddPonkaToImage` object, it *also* deserializes the `ImagePost` object that lives inside. At *that* moment, Doctrine's identity map does *not* contain this object... because it did not query for it inside this PHP process - from inside the worker. That's why originally, before we added `persist()`, when we called `flush()`, Doctrine looked at the list of objects in its identity map - which was *empty* - and... did absolutely nothing: it doesn't know it's supposed to save the `ImagePost`!

When we added `persist()`, we created a different issue. Doctrine *is* now aware that it needs to save this... but because it didn't original query for it, it mistakenly thinks that this should be *inserted* into the database as a *new* row, instead of updating.

Phew! I wanted you to see this because... it *is* kinda hard to debug. Fortunately, the fix is easy. *And* it touches on an important best-practice for your messages: include *only* the information you need. That's next.

# Chapter 8: Passing Entity Ids inside of Messages

Suppose you need your friend to come over and watch your dog for the weekend - let's call her Molly. So you write them a message explaining all the details they need to know: how often to feed Molly, when to walk her, exactly where she likes to be scratched behind the ears, your favorite superhero movie and the name of your childhood best friend. Wait... those last two things... while *fascinating*... have *nothing* to do with watching your dog Molly!

And this touches on a best-practice for designing your message classes: make them contain *all* the details the handler needs... and *nothing* extra. This isn't an absolute rule... it just makes them leaner, smaller and more directed.

## Passing the Entity Id

If you think about our message, we don't *really* need the entire `ImagePost` object. The *smallest* thing that we could pass is actually the id... which we could then use to query for the `ImagePost` object and get the filename.

Change the constructor argument to `int $imagePostId`. I'll change that below and go to Code -> Refactor to rename the property. Oh, and brilliant! It also renamed my getter to `getImagePostId()`. Update the return type to be an `int`. We can remove the old `use` statement as extra credit.

src/Message/AddPonkaToImage.php

```
↕ // ... lines 1 - 2
3  namespace App\Message;
4
5  use App\Entity\ImagePost;
6
7  class AddPonkaToImage
8  {
9      private $imagePostId;
10
11     public function __construct(int $imagePostId)
12     {
13         $this->imagePostId = $imagePostId;
14     }
15
16     public function getImagePostId(): int
17     {
18         return $this->imagePostId;
19     }
20 }
```

Next, in `ImagePostController`, search for `AddPonkaToImage` and... change this to `$imagePost->getId()`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 21
22 class ImagePostController extends AbstractController
23 {
↕ // ... lines 24 - 38
39     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
40     {
↕ // ... lines 41 - 60
61         $message = new AddPonkaToImage($imagePost->getId());
↕ // ... lines 62 - 64
65     }
↕ // ... lines 66 - 93
94 }
```

Our message class is now as *small* as it can get. Of course, this means that we have a *little* bit extra work to do in our handler. First, the `$imagePost` variable is not... well.. an `ImagePost` anymore! Rename it to `$imagePostId`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 11
12 class AddPonkaToImageHandler implements MessageHandlerInterface
13 {
↕ // ... lines 14 - 26
27     public function __invoke(AddPonkaToImage $addPonkaToImage)
28     {
29         $imagePostId = $addPonkaToImage->getImagePostId();
↕ // ... lines 30 - 37
38     }
39 }
```

To query for the actual object, add a new constructor argument: `ImagePostRepository $imagePostRepository`. I'll hit Alt + Enter -> Initialize Fields to create that property and set it.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 7
8  use App\Repository\ImagePostRepository;
↕ // ... lines 9 - 11
12 class AddPonkaToImageHandler implements MessageHandlerInterface
13 {
↕ // ... lines 14 - 16
17     private $imagePostRepository;
↕ // ... line 18
19     public function __construct(PhotoPonkaficator $ponkaficator,
       PhotoFileManager $photoManager, EntityManagerInterface $entityManager,
       ImagePostRepository $imagePostRepository)
20     {
↕ // ... lines 21 - 23
24         $this->imagePostRepository = $imagePostRepository;
25     }
↕ // ... lines 26 - 38
39 }
```

Back in the method, we can say `$imagePost = $this->imagePostRepository->find($imagePostId)`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 11
12 class AddPonkaToImageHandler implements MessageHandlerInterface
13 {
↕ // ... lines 14 - 26
27     public function __invoke(AddPonkaToImage $addPonkaToImage)
28     {
↕ // ... line 29
30         $imagePost = $this->imagePostRepository->find($imagePostId);
↕ // ... lines 31 - 37
38     }
39 }
```

That's it! And this fixes our Doctrine problem! Now that we're querying for the entity, when we call `flush()`, it will correctly save it with an `UPDATE`. We can remove the `persist()` call because it's not needed for updates.

Let's try it! Because we just changed code in our handler, hit Ctrl+C to stop our worker and then restart it:

```
php bin/console messenger:consume -vv
```

Here we go! Upload a new file... check the worker - yep, it processed just fine - and... refresh! Yes! *No* duplication, Ponka is visiting my workshop *and* the date is set!

## Failing Gracefully

But... sorry to bring up bad news... what if the `ImagePost` can't be found for this `$imagePostId`? That *shouldn't* happen... but depending on your app, it might be possible! For us... it is! If a user uploads a photo, then *deletes* it before the worker can handle it, the `ImagePost` will be gone!

Is that really a problem? If the `ImagePost` was already deleted, do we care if this handler blows up? Probably not... as long as you've thought about *how* it will explode and are intentional.

Check this out: let's start by saying: `if (!$imagePost)` so we can do some special handling... instead of trying to call `getFilename()` on null down here. If this happens, we

know that it's *probably* just because the image was already deleted. But... because I *hate* surprises on production, let's log a message so that we know this happened... *just* in case it's caused by a bug in our code.

## Logger Injection with LoggerAwareInterface

Starting in Symfony 4.2, there's a little shortcut to getting the main `logger` service. First, make your service implement `LoggerAwareInterface`. Then, use a trait called `LoggerAwareTrait`.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 9
10 use Psr\Log\LoggerAwareInterface;
11 use Psr\Log\LoggerAwareTrait;
↕ // ... lines 12 - 13
14 class AddPonkaToImageHandler implements MessageHandlerInterface,
   LoggerAwareInterface
15 {
16     use LoggerAwareTrait;
↕ // ... lines 17 - 30
31     public function __invoke(AddPonkaToImage $addPonkaToImage)
32     {
↕ // ... lines 33 - 35
36         if (!$imagePost) {
↕ // ... lines 37 - 44
45         }
↕ // ... lines 46 - 52
53     }
54 }
```

That's it! Let's peek inside `LoggerAwareTrait`. Ok cool. In the core of Symfony, there's a *little* bit of code that says:

> *"whenever you see a user's service that implements `LoggerAwareInterface`, automatically call `setLogger()` on it and pass the logger."*

By combining the interface with this trait... we don't have to do anything! We instantly have a `$logger` property we can use.

## How to Fail in your Handler

Ok, so back inside our if statement... what should we do if the `ImagePost` isn't found? We have two options... and the correct choice depends on the situation. First, we could throw an exception - any exception - and that would cause this message to be retried. More retries soon. Or, you could simply "return" and this message will "appear" to have been handled successfully... and will be removed from the queue.

Let's return: there's no point in retrying this message later... that `ImagePost` is *gone*!

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 13
14 class AddPonkaToImageHandler implements MessageHandlerInterface,
   LoggerAwareInterface
15 {
↕ // ... lines 16 - 30
31     public function __invoke(AddPonkaToImage $addPonkaToImage)
32     {
↕ // ... lines 33 - 35
36         if (!$imagePost) {
37             // could throw an exception... it would be retried
38             // or return and this message will be discarded
↕ // ... lines 39 - 43
44             return;
45         }
↕ // ... lines 46 - 52
53     }
54 }
```

But let's also log a message: if `$this->logger`, then `$this->logger->alert()` with, how about,

> *"Image post %d was missing!"*

passing `$imagePostId` for the wildcard.

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 13
14 class AddPonkaToImageHandler implements MessageHandlerInterface,
   LoggerAwareInterface
15 {
↕ // ... lines 16 - 30
31     public function __invoke(AddPonkaToImage $addPonkaToImage)
32     {
↕ // ... lines 33 - 35
36         if (!$imagePost) {
37             // could throw an exception... it would be retried
38             // or return and this message will be discarded
39
40             if ($this->logger) {
41                 $this->logger->alert(sprintf('Image post %d was missing!',
   $imagePostId));
42             }
43
44             return;
45         }
↕ // ... lines 46 - 52
53     }
54 }
```

Oh, and the only reason I'm checking to see *if* `$this->logger` is set is... basically... to help with unit testing. Inside Symfony, the `logger` property *will* always be set. But on an object-oriented level, there's nothing that *guarantees* that someone will have called `setLogger()`... so this is just a bit more responsible.

## Witnessing Errors in your Handler

Let's try this thang! Let's see what happens if we delete an `ImagePost` before it's processed! First, move over, stop the handler, and restart it:

```
php bin/console messenger:consume -vv
```

And because each message takes a few seconds to process, if we upload a *bunch* of photos... and delete them *super* quick... with any luck, we'll delete one *before* its message is handled.

Let's see if it worked! So... some *did* process successfully. But... yea! This one has an alert! And thanks to the "return" we added, it was "acknowledged"... meaning it was removed from the queue.

Oh... and interesting... there's another error I didn't plan for below:

> *“An exception occurred while handling message AddPonkaToImage: File not found at path...”*

That's awesome! *This* is what it looks like if, for *any* reason, an *exception* is thrown in your handler. Apparently the `ImagePost` *was* found in the database... but by the time it tried to read the file on the filesystem, it had been deleted!

The *really* amazing part is that Messenger saw this failure and *automatically* retried the message a second... then a third time. We'll talk more about failures and retries a bit later.

But first, our `DeleteImagePost` message *is* still being handled synchronously. Could we make it async? Well... no! We *need* the `ImagePost` to be deleted from the database immediately so that the user doesn't see it if they refresh. Unless... we could split the delete task into two pieces... Let's try that next!

# Chapter 9: Dispatching a Message inside a Handler?

Deleting an image is still done synchronously. You can see it: because I made it *extra* slow for dramatic effect, it takes a couple of seconds to process before it disappears. Of course, we could hack around this by making our JavaScript remove the image visually *before* the AJAX call finishes. But making heavy stuff *async* is a good practice and could allow us to put less load on the web server.

Let's look at the current state of things: we *did* update all of this to be handled by our command bus: we have a `DeleteImagePost` command and `DeleteImagePostHandler`. But inside `config/packages/messenger.yaml`, we're not routing this class anywhere, which means it's being handled immediately.

Oh, and notice: we're *still* passing the entire entity object into the message. In the last two chapters, we talked about avoiding this as a best practice *and* because it can cause weird things to happen if you handle this async.

But... if you're planning to keep `DeleteImagePost` synchronous... it's up to you: passing the entire entity object won't hurt anything. And... really... we *do* need this message to be handled synchronously! We need the `ImagePost` to be deleted from the database immediately so that, if the user refreshes, the image is gone.

But, look closer: deleting involves *two* steps: deleting a row in the database and removing the underlying image file. And... only that *first* step needs to happen right now. If we delete the file on the filesystem later... that's no big deal!

## Splitting into a new Command+Handler

To do *part* of the work sync and the other part async, my preferred approach is to split this into two commands.

Create a new command class called `DeletePhotoFile`. Inside, add a constructor so we can pass in whatever info we need. *This* command class will be used to *physically* remove the file

from the filesystem. And if you look in the handler, to do this, we only need the `PhotoFileManager` service and the string *filename*.

So this time, the *smallest* amount of info we can put in the command class is `string $filename`.

src/Message/DeletePhotoFile.php

```
↕ // ... lines 1 - 2
3  namespace App\Message;
4
5  class DeletePhotoFile
6  {
↕ // ... lines 7 - 8
9      public function __construct(string $filename)
10     {
↕ // ... line 11
12     }
↕ // ... lines 13 - 17
18 }
```

I'll hit Alt + enter and go to "Initialize Fields" to create that property and set it.

src/Message/DeletePhotoFile.php

```
↕ // ... lines 1 - 2
3  namespace App\Message;
4
5  class DeletePhotoFile
6  {
7      private $filename;
8
9      public function __construct(string $filename)
10     {
11         $this->filename = $filename;
12     }
↕ // ... lines 13 - 17
18 }
```

Now I'll go to Code -> Generate - or Cmd+N on a Mac - to generate the getter.

src/Message/DeletePhotoFile.php

```
↕ // ... lines 1 - 2
3  namespace App\Message;
4
5  class DeletePhotoFile
6  {
7      private $filename;
8
9      public function __construct(string $filename)
10     {
11         $this->filename = $filename;
12     }
13
14     public function getFilename(): string
15     {
16         return $this->filename;
17     }
18 }
```

Cool! Step 2: add the handler `DeletePhotoFileHandler`. Make this follow the two rules for handlers: implement `MessageHandlerInterface` and create an `__invoke()` method with one argument that's type-hinted with the message class: `DeletePhotoFile $deletePhotoFile`.

src/MessageHandler/DeletePhotoFileHandler.php

```
↕ // ... lines 1 - 2
3  namespace App\MessageHandler;
4
5  use App\Message\DeletePhotoFile;
↕ // ... line 6
7  use Symfony\Component\Messenger\Handler\MessageHandlerInterface;
8
9  class DeletePhotoFileHandler implements MessageHandlerInterface
10 {
↕ // ... lines 11 - 17
18     public function __invoke(DeletePhotoFile $deletePhotoFile)
19     {
↕ // ... line 20
21     }
22 }
```

Perfect! The *only* thing we need to do in here is... this one line: `$this->photoManager->deleteImage()`. Copy that and paste it into our handler. For the argument, we can use our message class: `$deletePhotoFile->getFilename()`.

src/MessageHandler/DeletePhotoFileHandler.php

```
// ... lines 1 - 2
namespace App\MessageHandler;

use App\Message\DeletePhotoFile;
// ... line 6
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class DeletePhotoFileHandler implements MessageHandlerInterface
{
// ... lines 11 - 17
    public function __invoke(DeletePhotoFile $deletePhotoFile)
    {
        $this->photoManager->deleteImage($deletePhotoFile->getFilename());
    }
}
```

And finally, we need the `PhotoFileManager` service: add a constructor with one argument: `PhotoFileManager $photoManager`. I'll use my Alt+Enter -> Initialize fields trick to create that property as usual.

src/MessageHandler/DeletePhotoFileHandler.php

```
// ... lines 1 - 2
namespace App\MessageHandler;

use App\Message\DeletePhotoFile;
use App\Photo\PhotoFileManager;
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class DeletePhotoFileHandler implements MessageHandlerInterface
{
    private $photoManager;

    public function __construct(PhotoFileManager $photoManager)
    {
        $this->photoManager = $photoManager;
    }

    public function __invoke(DeletePhotoFile $deletePhotoFile)
    {
        $this->photoManager->deleteImage($deletePhotoFile->getFilename());
    }
}
```

Done! We now have a functional command class which requires the string filename, and a handler that *reads* that filename and... does the work!

## Dispatching Embedded

All we need to do now is *dispatch* the new command. And... *technically* we could do this in two different places. First, you might be thinking that, in `ImagePostController`, we could dispatch two different commands right here.

But... I don't *love* that. The controller is already saying `DeleteImagePost`. It shouldn't need to issue any other commands. If we choose to break that logic down into *smaller* pieces, that's up to the handler. In other words, we're going to dispatch this new command from *within* the command handler. Inception!

Instead of calling `$this->photoManager->deleteImage()` directly, change the type-hint on that argument to autowire `MessageBusInterface $messageBus`. Update the code in the constructor... and the property name.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Messenger\MessageBusInterface;
↕ // ... line 11
12 class DeleteImagePostHandler implements MessageHandlerInterface
13 {
14     private $messageBus;
↕ // ... lines 15 - 16
17     public function __construct(MessageBusInterface $messageBus,
   EntityManagerInterface $entityManager)
18     {
19         $this->messageBus = $messageBus;
20         $this->entityManager = $entityManager;
21     }
↕ // ... lines 22 - 32
33 }
```

Now, easy: remove the old code and start with: `$filename = $imagePost->getFilename()`. Then, let's delete it from the database and, at the bottom, `$this->messageBus->dispatch(new DeletePhotoFile($filename))`.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 5
6 use App\Message\DeletePhotoFile;
↕ // ... lines 7 - 11
12 class DeleteImagePostHandler implements MessageHandlerInterface
13 {
↕ // ... lines 14 - 22
23     public function __invoke(DeleteImagePost $deleteImagePost)
24     {
25         $imagePost = $deleteImagePost->getImagePost();
26         $filename = $imagePost->getFilename();
27 
28         $this->entityManager->remove($imagePost);
29         $this->entityManager->flush();
30 
31         $this->messageBus->dispatch(new DeletePhotoFile($filename));
32     }
33 }
```

And... this should... just work: everything is *still* being handled synchronously.

Let's try it next, think a bit about what happens if *part* of a handler fails, and make half of the delete process async.

# Chapter 10: Partial Handler Failures & Advanced Routing

We just broke our image deleting process into smaller pieces by creating a *new* command class, a new handler and *dispatching* that new command from *within* the handler! This... technically isn't anything special, but it *is* cool to see how you can break each task down into as small pieces as you need.

But let's... make sure this actually works. Everything *should* still process synchronously. Delete the first image and... refresh to be sure. It's gone!

## Thinking about Failures and if Messages are Dispatched

Before we handle the new command class asynchronously, we need to think about something. If, for some reason, there's a problem removing this `ImagePost` from the database, Doctrine will throw an exception right here and the file will never be deleted. That's perfect: the row in the database and file on the filesystem will *both* remain.

But if deleting the row from the database is successful... but there's a problem deleting the file from the filesystem - like a temporary connection problem talking to S3 if our file were stored there... that file would... actually.. *never* be deleted! And... maybe you don't care. But if you *do*, you could wrap this *entire* block in a Doctrine transaction to make sure it's *all* successful before *finally* removing the row. Of course... once we change this message to be handled asynchronously, deleting the actual file will be done *later*... and we will be, kinda "trusting" that it will be handled successfully. We're going to talk about failures and retries *really* soon.

## Routing the Message Async

Anyways, now that we've broken this into two pieces, head over to `config/packages/messenger.yaml`. Copy the existing line, paste and route the new `DeletePhotoFile` to `async`.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 12
        routing:
// ... lines 14 - 15
            'App\Message\DeletePhotoFile': async
```

Cool! With any luck, the row in the database will be deleted immediately... then the *file* a few seconds later.

And because we just made a change to some handler code, go over, stop our worker and restart it:

```
php bin/console messenger:consume -vv
```

Testing time! Refresh to be safe... and let's try deleting. Check out how much faster that is! If you scoot over to the worker terminal... yea, it's doing all *kinds* of good stuff here. Oh, and fun! An exception occurred while handling one of the messages - a file wasn't found. I think that's from the duplicate row caused by the Doctrine bug a few minutes ago: the file was already gone when the second image was deleted. The cool thing is that it's already retrying that message in case it was a temporary failure. Eventually, it gives up and "rejects" the message.

Let's try this *whole* crazy system together! Upload a bunch of photos... then... quick! Delete a couple! If you look at the worker... it's all beautifully mixed up: a few `AddPonkaToImage` objects are handled here... then `DeletePhotoFile`.

## Routing with Interfaces & Base Classes

Oh, and by the way: if you look at the `routing` section in `messenger.yaml`, you'll *usually* route thing by their exact class name: `App\Message\AddPonkaToImage` goes to `async`. But you can *also* route via interfaces or base classes. For example, if you have a *bunch* of classes that should go to the `async` transport, you *could* create your very own interface - maybe `AsyncMessageInterface` - make your messages implement that, then *only* need to route that interface to `async` here. But be careful because, if a class matches *multiple* routing lines, it *will* be sent to *all* those transports. Oh, and last thing - in case you have a use-case, each routing entry can send to *multiple* transports.

Next: remember how the serialized message in the database was wrapped in something called an `Envelope`? Let's learn what that is and how its *stamp* system gives us some cool superpowers.

# Chapter 11: Envelopes & Stamps

We just got a request from Ponka herself... and when it comes to this site, Ponka is the boss. She thinks that, when a user uploads a photo, her image is actually being added a little bit *too* quickly. She wants it to take longer: she wants it to feel like she's doing some *really* epic work behind the scenes to get into your photo.

I know, it's *kind* of a silly example - Ponka is so weird when you talk to her before her shrimp breakfast and morning nap . But... it *is* an interesting challenge: could we somehow not *only* say: "handle this later"... but also "wait at least 5 seconds before handling it?".

## Envelope: A Great Place to put a Message

Yep! And it touches on some super cool parts of the system called stamps and envelopes. First, open up `ImagePostController` and go up to where we create the `AddPonkaToImage` object. `AddPonkaToImage` is called the "message" - we know that. What we *don't* know is that, when you pass your message to the bus, internally, it gets wrapped inside something called an `Envelope`.

Now, this isn't an especially important detail except that, if you have an `Envelope`, you can attach extra config to it via *stamps*. So yes, you literally put a message in an envelope and then attach stamps. Is this your favorite component or what?

Anyways, those stamps can carry all sorts of info. For example, if you're using RabbitMQ, you can configure a few things about how the message is *delivered*, like something called a "routing key". Or, you can configure a delay.

## Put the Message into the Envelope, then add Stamps

Check this out: say `$envelope = new Envelope()` and pass it our `$message`. Then, pass this an optional second argument: an array of stamps.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 15
16 use Symfony\Component\Messenger\Envelope;
↕ // ... lines 17 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
↕ // ... line 65
66         ]);
↕ // ... lines 67 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Include just one: `new DelayStamp(5000)`. This indicates to the transport... which is kind of like the mail carrier... that you'd like this message to be delayed 5 seconds before it's delivered. Finally, pass the `$envelope` - *not* the message - into `$messageBus->dispatch()`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 17
18 use Symfony\Component\Messenger\Stamp\DelayStamp;
↕ // ... lines 19 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             new DelayStamp(5000)
66         ]);
67         $messageBus->dispatch($envelope);
↕ // ... lines 68 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Yep, the `dispatch()` method accepts raw message objects *or* `Envelope` objects. If you pass a raw message, it wraps it in an `Envelope`. If you *do* pass an `Envelope`, it uses it! The end result is the same as before... except that we're now applying a `DelayStamp`.

Let's try it! This time we *don't* need to restart our worker because we haven't changed any code *it* will use: we only changed code that controls how the message will be *delivered*. But... if you're ever not sure - just restart it.

I *will* clear the console so we can watch what happens. Then... let's upload three photos and... one, two, three, four there it is! It delayed 5 seconds and *then* started processing each like normal. There's not a 5 second delay *between* handling each message: it just makes sure that each message is handled no *sooner* than 5 seconds after sending it.

> **Tip**
>
> Support for delays in Redis WAS added in Symfony 4.4.

Side note: In Symfony 4.3, the Redis transport doesn't support delays - but it may be added in the future.

## What other Stamps are There?

Anyways, you may not use stamps a *ton*, but you will need them from time-to-time. You'll probably Google "How do I configure validation groups in Messenger" and learn *which* stamp controls this. Don't worry, I'll talk about validation later - it's *not* something that's happening right now.

One *other* cool thing is that, internally, Messenger *itself* uses stamps to track and help deliver messages correctly. Check this out: wrap `$messageBus->dispatch()` in a `dump()` call.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
$validator, PhotoFileManager $photoManager, EntityManagerInterface
$entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 66
67         dump($messageBus->dispatch($envelope));
↕ // ... lines 68 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Let's go over and upload one new image. Then, on the web debug toolbar, find the AJAX request that just finished - it'll be the bottom one - click to open its profiler and then click "Debug" on the left. There it is! The `dispatch()` method *returns* an `Envelope`... which holds the message of course... and *now* has *four* stamps! It has the `DelayStamp` like we expected, but also a `BusNameStamp`, which records the name of the bus that it was sent to. This is cool: we only have one bus now, but you're allowed to have *multiple*, and we'll talk about why you might do that later. The `BusNameStamp` helps the worker command know *which* bus to send the `Envelope` to after it's read from the transport.

That `SentStamp` is basically a marker that says "this message was sent to a transport instead of being handled immediately" and this `TransportMessageIdStamp`, literally contains the *id* of the new row in the `messenger_messages` table... in case that's useful.

You don't *really* need to care about any of this - but watching what stamps are being added to your `Envelope` may help you debug an issue or do some more advanced stuff. In fact, some of these will come in handy soon when we talk about middleware.

For now, remove the `dump()` and then, so I don't drive myself crazy with how slow this is, change the `DelayStamp` to 500 milliseconds. Shh, don't tell Ponka. After this change... yep! The message is handled *almost* immediately.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             new DelayStamp(500)
66         ]);
67         $messageBus->dispatch($envelope);
↕ // ... lines 68 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Next, let's talk about retries and what happens when things go wrong! No joke: this stuff is *super* cool.

# Chapter 12: Retrying on Failure

When you start handling things asynchronously, thinking about what happens when code fails is even *more* important! Why? Well, when you handle things synchronously, if something fails, typically, the *whole* process fails, not just half of it. Or, at least, you *can* make the whole process fail if you need to.

For example: pretend all our code is still synchronous: we save the `ImagePost` to the database, but then, down here, adding Ponka to the image fails... because she's napping. Right now, that *would* result in *half* of the work being done... which, depending on how sensitive your app is, may or may not be a huge deal. If it *is*, you can solve it by wrapping all of this in a database transaction.

Thinking about *how* things will fail - and coding defensively when you need to - is just a healthy programming practice.

## The Difficulty of Async Failures

But this all changes when some code is async! Think about it: we save the `ImagePost` to the database, `AddPonkaToImage` is sent to the transport and the response is successfully returned. Then, a few seconds later, our worker processes that message and, due to a temporary network problem, the handler throws an exception!

That's... not a great situation. The user thinks everything was ok because they didn't see an error. And now we have an `ImagePost` in the database... but Ponka will *never* be added to it. Ponka is furious.

The point is: when you send a message to a transport, we *really* need to make sure that the message *is* eventually processed. If it's not, it could lead to some weird conditions in our system.

## Watching Failures

So let's start making our code fail to see what happens! Inside `AddPonkaToImageHandler`, right before the real code runs, say if `rand(0, 10) < 7`, then throw a `new \Exception()` with:

> *"I failed randomly!!!!"*

src/MessageHandler/AddPonkaToImageHandler.php

```
↕ // ... lines 1 - 13
14 class AddPonkaToImageHandler implements MessageHandlerInterface,
   LoggerAwareInterface
15 {
↕ // ... lines 16 - 30
31     public function __invoke(AddPonkaToImage $addPonkaToImage)
32     {
↕ // ... lines 33 - 46
47         if (rand(0, 10) < 7) {
48             throw new \Exception('I failed randomly!!!!');
49         }
↕ // ... lines 50 - 56
57     }
58 }
```

Let's see what happens! First, go restart the worker:

```
php bin/console messenger:consume -vv
```

Then I'll clear the screen and... let's upload! How about five photos? Go back over to see what's happening! Whoa! A *lot* is happening. Let's pull this apart.

The first message was received and handled. The second message was received and *also* handled successfully. The third message was received but an exception occurred while handling it: "I failed randomly!". Then it says: "Retrying - retry #1" followed by "Sending message". Yea, because it failed, Messenger *automatically* "retries" it... which *literally* means that it sends that message *back* to the queue to be processed later! One of these "Received message" logs down here is *actually* that message being received for a *second* time, thanks to the retry. The cool thing is... eventually... all the messages *were* handled successfully! That's why retries rock. We can see this when we refresh: *everyone* has a Ponka photo... even though some of these failed at first.

## Hitting the 3 Retry Max

But... let's try this again... because that example didn't show the *most* interesting case. I'll select *all* the photos this time... oh, but first, let's clear the screen on our worker terminal. Ok, upload, then... move over.

Here we go: this time... thanks to randomness, we're seeing a *lot* more failures. We see that a couple of messages failed and were sent for retry #1. Then, some of those messages failed *again* and were sent for retry #2! And... yea! They failed yet *again* and were sent for retry #3. Finally... oh yes, perfect: after being attempted once and retried again 3 *more* times, one of the messages *still* failed. This time, instead of sending for retry #4, it says:

> *"Rejecting AddPonkaToImage (removing from transport)"*

Here's what's going on: by default, Messenger will retry a message three times. If it *still* fails, it's finally removed from the transport and the message is lost permanently. Well... that's not *totally* true... and there's a bit more going on here than it seems at first.

Next, if you look closely... these retries are *delayed* at an increasing level. Let's learn why and how to take *complete* control over how your messages are retried.

# Chapter 13: Retry Delay & Retry Strategy

By default, a message will be retried three times then lost forever. Well... in a few minutes... I'll show you how you can *avoid* even *those* messages from being lost.

Anyways... the process... just works! And it's even cooler than it looks at first. It's a bit hard to see - especially because there's a sleep in our handler - but this message was sent for retry #3 at the 13 second timestamp and it was finally handled again down at the 17 second timestamp - a 4 second delay. That delay was *not* caused by our worker just being busy until then: it was 100% intentional.

Check it out: I'll hit Ctrl+C to stop the worker and then run:

```
php bin/console config:dump framework messenger
```

This should give us a big tree of "example" configuration that you can put under the `framework` `messenger` config key. I *love* this command: it's a *great* way to find options that you maybe didn't know existed.

Cool! Look closely at the `transports` key - it lists an "example" transport below with *all* the possible config options. One of them is `retry_strategy` where we can control the maximum number of retries and the *delay* that should happen between those retries.

This `delay` number is smarter than it looks: it works together with the "multiplier" to create an exponentially growing delay. With these settings, the first retry will delay one second, the second 2 seconds and the third *4* seconds.

This is important because, if a message fails due to some temporary issue - like connecting to a third-party server - you might *not* want to try again immediately. In fact, you might choose to set these to *way* higher values so that it retries maybe 1 minute or even a day later.

Let's also try a similar command:

```
php bin/console debug:config framework messenger
```

Instead of showing *example* config, this tells us what our *current* configuration is, including any default values: our `async` transport has a `retry_strategy`, which is defaulting to 3 max retries with a 1000 millisecond delay and a multiplier of 2.

## Configuring the Delay

Let's make this a bit *more* interesting. In the handler, let's make it *always* fail by adding `|| true`.

src/MessageHandler/AddPonkaToImageHandler.php

```
// ... lines 1 - 13
class AddPonkaToImageHandler implements MessageHandlerInterface, LoggerAwareInterface
{
// ... lines 16 - 30
    public function __invoke(AddPonkaToImage $addPonkaToImage)
    {
// ... lines 33 - 46
        if (rand(0, 10) < 7 || true) {
            throw new \Exception('I failed randomly!!!!');
        }
// ... lines 50 - 56
    }
}
```

Now, under `messenger`, let's play with the retry config. Wait... but the `async` transport is set to a string... are we allowed to include config options under that? No! Well, yes, sort of. As soon as you need to configure a transport beyond just the connection details, you'll need to drop this string onto the next line and assign it to a `dsn` key. *Now* we can add `retry_strategy`, and let's set the delay to 2 seconds instead of 1.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 5
        transports:
            # https://symfony.com/doc/current/messenger.html#transports
            async:
                dsn: '%env(MESSENGER_TRANSPORT_DSN)%'
                retry_strategy:
                    delay: 2000
// ... lines 12 - 20
```

Oh, and I also want to mention this `service` key. If you want to *completely* control the retry config - maybe even having different retry logic per message - you can create a service that implements `RetryStrategyInterface` and put its service id - usually its class name - right here.

Anyways, let's see what happens with the longer delay: restart the worker process:

```
php bin/console messenger:consume -vv
```

This time, upload just *one* photo so we can watch it fail over and over again. And... yep! It fails and sends for retry #1... then fails again and sends for retry #2. But check out that delay! 09 to 11 - 2 seconds - then 11 to 15 - a 4 second delay. And... if... we... are... super... patient... yea! Retry #3 starts a full *8* seconds later. Then it's "rejected" - removed from the queue - and lost forever. Tragic!

Retries are great... but I don't like that last part: when the message is eventually lost forever. Change the delay to 500 - it'll make this easier to test.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 5
        transports:
            # https://symfony.com/doc/current/messenger.html#transports
            async:
                dsn: '%env(MESSENGER_TRANSPORT_DSN)%'
                retry_strategy:
                    delay: 500
// ... lines 12 - 20
```

Next, let's talk about a special concept called the "failure transport": a better alternative than allowing failed messages to simply... disappear.

# Chapter 14: The Failure Transport

We now know that each message will be retried 3 times - which is configurable - and then, if handling it *still* fails, it will be "rejected"... which is a "queue" word for: it will be removed from the transport and lost forever.

That's... a bummer! Our last retry happened 14 seconds after our first... but if the handler is failing because a third-party server is temporarily down... then if that server is down for even just *30* seconds... the message will be lost forever! It would be better if we could retry it once the server was back up!

The answer to this is... the failure transport!

## Hello Failure Transport

First, I'm going to uncomment a *second* transport. In general, you can have as *many* transports as you want. This one starts with `doctrine://default`. If you look at our `.env` file... hey! That's *exactly* what our `MESSENGER_TRANSPORT_DSN` environment variable is set to! Yep, *both* our `async` and new `failed` transports are using the `doctrine` transport and the `default` doctrine connection. But the second one *also* has this little `?queue_name=failed` option. OooooOOOOooo.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 5
        transports:
// ... lines 7 - 12
            failed: 'doctrine://default?queue_name=failed'
// ... lines 14 - 20
```

Go back to whatever you're using to inspect the database and check out the queue table:

```
DESCRIBE messenger_messages;
```

Ah. One of the columns in this table is called `queue_name`. This column allows us to create *multiple* transports that all store messages in the same table. Messenger knows *which* messages belong to which transport thanks to this value. All the messages sent to the `failed` transport will have a `failed` value... that could be anything - and messages sent to the `async` transport will use the default value... which is `default`.

## Configuring Transports

By the way, each transport has a *number* of different connection options and there are two ways to pass them: either as query parameters like this *or* via an expanded format where you put the `dsn` on its own line and then add an `options` key with whatever you need below that.

What options can you put here? Each transport *type* - like `doctrine` or `amqp` - has its *own* set of options. Right now, they're not well-documented, but they *are* easy to find... if you know where to look. By convention, every transport type has a class called `Connection`. I'll press Shift+Shift in PhpStorm, search for `Connection.php`... and look for files. There they are! A `Connection` class for Amqp, Doctrine and Redis.

Open the one for Doctrine. All of these classes have documentation near the top that describe their options, in this case: `queue_name`, `table_name` and a few others, including `auto_setup`. Earlier, we saw that Doctrine will create the `messenger_messages` table automatically if it doesn't exist. If you don't want that to happen, you would set `auto_setup` to `false`.

The transport with the *most* options can be seen in the Amqp Connection class. We'll talk about Amqp later in the tutorial.

## The failure_transport

Anyways, back to it! We now have a new transport called `failed`... which, despite its name, is the same as any other transport. If we wanted to, we could *route* message classes there and consume them, *just* like we're doing for `async`.

But... the *purpose* of this transport is different. Near the top, there's another key called `failure_transport`. Uncomment that and notice that this *points* to our new `failed` transport.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
        # Uncomment this (and the failed transport below) to send failed messages to this transport for later handling.
        failure_transport: failed
// ... lines 5 - 20
```

What does it do? Let's see it in action! First, go restart our worker:

```
php bin/console messenger:consume -vv
```

Woh! This time, it *asks* us which "receiver" - which basically means which "transport" - we want to consume. A worker can read from one *or* many transports - something we'll talk about later with "prioritized" transports. Let's consume just the `async` transport - we'll handle messages from the `failed` transport in a different way. And actually, to make life easier, we can pass `async` as an argument so that it won't ask us which transport to use each time:

```
php bin/console messenger:consume -vv async
```

Now... let's upload some images! Then... over here... pretty quickly, all 4 of those exhaust their retries and are eventually rejected from the transport. Until now, that meant that they were gone forever. But this time... that did *not* happen. Before removing the message from the queue, it says:

> *"Rejected message `AddPonkaToImage` will be sent to the failure transport "failed""*

And then... "Sending message". So, it *was* removed from the `async` transport, but it still exists because it was sent to the "failed" transport.

How can we see what messages have failed and try them again if we think those failure were temporary? With a couple of shiny, new console commands. Let's talk about those next.

# Chapter 15: Investigating & Retrying Failed Messages

Apparently now that we've configured a `failure_transport`, if handling a message *still* isn't working after 3 retries, instead of being sent to `/dev/null`, they're sent to another transport - in our case called "failed". That transport is... really... the same as *any* other transport... and we *could* use the `messenger:consume` command to try to process those messages again.

But, there's a better way. Run:

```
php bin/console messenger
```

## Seeing Messages on the Failed Queue

Hey! Shiny new commands are hiding here! *Three* under `messenger:failed`. Try out that `messenger:failed:show` one:

```
php bin/console messenger:failed:show
```

Nice! *There* are our *4* failed messages... just sitting there wait for us to look at them. Let's pretend that we're not sure what went wrong with these messages and want to check them out. Start by passing the 115 id:

```
php bin/console messenger:failed:show 115
```

I love this: it shows us the error message, error class and a history of the message's misadventures through our system! It failed, was redelivered to the async transport at 05, at 06 and then at 07, it finally failed and was redelivered to the `failed` transport.

If we add a `-vv` on the command...

```
php bin/console messenger:failed:show 115 -vv
```

*Now* we can see a full stack trace of what happened on that exception.

This is a really powerful way to figure out what went wrong and what to do next: do we have a bug in our app that we need to fix before retrying this? Or maybe it was a temporary failure and we can try again now? Or maybe, for some reason, we just want to remove this message entirely.

If you *did* want to remove this without retrying, that's the `messenger:failed:remove` command.

## Retrying Failed Messages

But... let's retry this! Back in the handler, change this back to fail randomly.

src/MessageHandler/AddPonkaToImageHandler.php

```
// ... lines 1 - 13
class AddPonkaToImageHandler implements MessageHandlerInterface, LoggerAwareInterface
{
// ... lines 16 - 30
    public function __invoke(AddPonkaToImage $addPonkaToImage)
    {
// ... lines 33 - 46
        if (rand(0, 10) < 7) {
            throw new \Exception('I failed randomly!!!!');
        }
// ... lines 50 - 56
    }
}
```

There are two ways to work with the retry command: you can retry a specific id like you see here *or* you can retry the messages one-by-one. Let's do that. Run:

```
php bin/console messenger:failed:retry
```

This is kind of similar to how `messenger:consume` works, except that it asks you before trying each message and, instead of running this command *all* the time on production, you'll run it manually whenever you have some failed messages that you need to process.

Cool! We see the details and it asks if we want to retry this. Like with show, you can pass `-vv` to see the *full* message details. Say "yes". It processes... and then continues to the next. Actually, let me try that again with `-vv` so we can see what's going on:

```
php bin/console messenger:failed:retry -vv
```

## When Failed Messages... Fail Again

*This* time we see *all* the details. Say "yes" again and... nice: "Received message", "Message handled" and onto the *next* message. We're on a roll! Notice that this message's id is 117 - that'll be important in a second. Hit yes to retry this message too.

Woh! This time it failed again! What does that mean? Well remember, the failure transport is *really* just a normal transport that we're using in a special way. And so, when a message fails here, Messenger... retries it! Yea it was sent *back* to the failure transport!

I'll hit Control+C and re-run the show command:

```
php bin/console messenger:failed:show
```

That id 119 was *not* there when we started. Nope, when message 117 was processed, it failed, was *redelivered* to the failure transport as id 119, and *then* was removed. And so, unless you change your configuration, messages will be retried 3 times on the failure transport before *finally* being *completely* discarded.

Oh, but if you look at the retried message closer:

```
php bin/console messenger:failed:show 119 -vv
```

There's a bit of a bug: the error and error class are missing. The data *is* still in the database... it's just not displayed correctly here. But you *can* see the message's history: including that it was sent to the `failed` transport and then sent *again* to the `failed` transport.

By the way, you can pass a `--force` option to the `retry` command if you want it to retry messages one-by-one *without* asking you each time whether or not it should do it. Also, not *all* the transport types - like AMQP or Redis - support *all* of the features we just saw if you use it as your failure transport. That may change in the future, but at this moment - Doctrine is the *most* robust transport to use for failures.

Anyways, as cool as failing is, let's go back and remove the code that's breaking our handler. Because... it's time to take a step deeper into how Messenger works: it's time to talk about middleware.

# Chapter 16: Middleware

Internally, when you dispatch a message onto the bus... what happens? What does the code look like inside the bus? The answer is... there basically *is* no code inside the bus! *Everything* is done via middleware.

## Middleware Basics

The bus is nothing more than a collection of "middleware". And each middleware is just a function that receives the message and can do something with it.

The process looks like this. We pass a message to the `dispatch()` method, then the bus passes *that* to the first middleware. The middleware then runs some code and eventually calls the *second* middleware. It runs some code and eventually calls the *third* middleware... until finally the last middleware - let's say it's the fourth middleware - has no one else to call. At that moment, the fourth middleware function finishes, then the third middleware function finishes, then the second, then the first. Thanks to this design, each middleware can run code *before* calling the next middleware or *after*.

This "middleware" concept isn't unique to Messenger or even PHP - it's a pattern. It can be both super useful... and a bit confusing... as it's a big circle. The point is this: with Messenger, if you want to hook into the dispatch process - like to log what's happening - you'll do that with a middleware. Heck, even the core functionality of messenger - executing handlers and sending messages to transports - is done with middleware! Those are called `HandleMessageMiddleware` and `SendMessageMiddleware` if you want to geek out and see how they work.

So here's our goal: each time we dispatch a message... from *anywhere*, I want to attach a unique id to that message and then use that to log what's happening over time to the message: when it's initially dispatched, when it's sent to the transport, and when it's *received* from the transport and handled. Heck, you could even use this to track how *long* an individual message took before it was processed or how many times it was retried.

## Creating a Middleware

*Creating* a middleware is actually fairly simple. Create a new directory inside `src/` called `Messenger/`... though... like with pretty much *everything* in Symfony, this directory could be called anything. Inside, add a class called, how about, `AuditMiddleware`.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 2
namespace App\Messenger;
// ... lines 4 - 5
use Symfony\Component\Messenger\Middleware\MiddlewareInterface;
// ... lines 7 - 8
class AuditMiddleware implements MiddlewareInterface
{
// ... lines 11 - 14
}
```

The only rule for middleware is that they must implement - surprise! - `MiddlewareInterface`. I'll go to "Code -> Generate" - or Command+N on a Mac - and select "Implement Methods". This interface requires just one: `handle()`. We'll talk about the "stack" thing in a second... but mostly... the signature of this method makes sense: we receive the `Envelope` and *return* an `Envelope`.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 2
namespace App\Messenger;

use Symfony\Component\Messenger\Envelope;
use Symfony\Component\Messenger\Middleware\MiddlewareInterface;
use Symfony\Component\Messenger\Middleware\StackInterface;

class AuditMiddleware implements MiddlewareInterface
{
    public function handle(Envelope $envelope, StackInterface $stack):
Envelope
    {
        // TODO
    }
}
```

The one line that your middleware will almost definitely need is this: `return $stack->next()->handle($envelope, $stack)`.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 7
8
9  class AuditMiddleware implements MiddlewareInterface
10 {
11     public function handle(Envelope $envelope, StackInterface $stack):
   Envelope
12     {
↕ // ... line 13
14         return $stack->next()->handle($envelope, $stack);
15     }
16 }
```

*This* is the line that basically says:

> *"I want to execute the next middleware and then return its value."*

Without this line, any middleware after us would *never* be called... which isn't *usually* what you want.

## Registering the Middleware

And... to start... that's enough: this class is already a functional middleware! But, unlike a lot of stuff in Symfony, Messenger won't find and start using this middleware automatically. Find your open terminal and, once again, run:

```
php bin/console debug:config framework messenger
```

Let's see... somewhere in here is a key called `buses`. This defines all of the message bus services you have in your system. Right now, we have one: the default bus called `messenger.bus.default`. That name could be anything and becomes the service id. Below this, we can use the `middleware` key to define whatever *new* middleware we want to add, in addition to the core ones that are added by default.

Let's copy that config. Then, open `config/packages/messenger.yaml` and, under `framework:`, `messenger:`, paste this right on top... and make sure it's indented correctly. Below, add `middleware:` a new line, then our new middleware service: `App\Messenger\AuditMiddleware`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
        buses:
            messenger.bus.default:
                middleware:
                    - App\Messenger\AuditMiddleware
// ... lines 7 - 25
```

## Order of Middleware

And just like that, our middleware *should* be called... along with all the *core* middleware. What... um... *are* the core middleware? And what order is everything called in? Well, there's not a great way to see that yet, but you *can* find this information by running:

```
php bin/console debug:container --show-arguments messenger.bus.default.inner
```

... which is a *super* low-level way to get information about the message bus. Anyways, there are a few core middleware at the start that get some basic things set up, then *our* middleware, and finally, `SendMessageMiddleware` and `HandleMessageMiddleware` are called at the end. Knowing the exact order of this stuff isn't that important - but hopefully it'll help demystify things as we keep going.

Next, let's get to work by using our middleware to attach a unique id to each message. How? Via our very own stamp!

# Chapter 17: Tracking Messages with Middleware & a Stamp

We somehow want to attach a unique id - just some string - that stays with the message forever: whether it's handled immediately, sent to a transport, or even retried multiple times.

## Creating a Stamp

How can we attach extra... "stuff" to a message? By giving it our very-own stamp! In the `Messenger/` directory, create a new PHP class called `UniqueIdStamp`. Stamps *also* have just one rule: they implement `MessengerEnvelopeMetadataAwareContainerReaderInterface`. Nah I'm kidding - that would be a silly name. They just need to implement `StampInterface`.

src/Messenger/UniqueIdStamp.php

```
↕ // ... lines 1 - 2
3 namespace App\Messenger;
4
5 use Symfony\Component\Messenger\Stamp\StampInterface;
6
7 class UniqueIdStamp implements StampInterface
8 {
↕ // ... lines 9 - 19
20 }
```

And... that's it! This is an empty interface that just serves to "mark" objects as stamps. Inside... we get to do *whatever* we want... as long as PHP can serialize this message... which basically means: as long as it holds simple data. Let's add a `private $uniqueId` property, then a constructor with no arguments. Inside, say `$this->uniqueId = uniqid()`. At the bottom, go to Code -> Generate - or Command+N on a Mac - and generate the getter... which will return a `string`.

src/Messenger/UniqueIdStamp.php

```php
// ... lines 1 - 2
namespace App\Messenger;

use Symfony\Component\Messenger\Stamp\StampInterface;

class UniqueIdStamp implements StampInterface
{
    private $uniqueId;

    public function __construct()
    {
        $this->uniqueId = uniqid();
    }

    public function getUniqueId(): string
    {
        return $this->uniqueId;
    }
}
```

Stamp, done!

## Stamping... um... Attaching the Stamp

Next, inside `AuditMiddleware`, *before* we call the next middleware - which will call the rest of the middleware and ultimately handle or send the message - let's add the stamp.

But, be careful: we need to make sure that we only attach the stamp *once*. As we'll see in a minute, a message may be passed to the bus - and so, to the middleware - *many* times! Once when it's initially dispatched and *again* when it's received from the transport and handled. If handling that message fails and is retried, it would go through the bus even *more* times.

So, start by checking if `null === $envelope->last(UniqueIdStamp::class)`, then `$envelope = $envelope->with(new UniqueIdStamp())`.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 8
9  class AuditMiddleware implements MiddlewareInterface
10 {
11     public function handle(Envelope $envelope, StackInterface $stack):
   Envelope
12     {
13         if (null === $envelope->last(UniqueIdStamp::class)) {
14             $envelope = $envelope->with(new UniqueIdStamp());
15         }
↕ // ... lines 16 - 21
22     }
23 }
```

## Envelopes are Immutable

There are a few interesting things here. First, each `Envelope` is "immutable", which means that, just due to the way that class was written, you can't change any data on it. When you call `$envelope->with()` to *add* a new stamp, it doesn't *actually* modify the `Envelope`. Nope, internally, it makes a clone of itself *plus* the new stamp.

That's... not very important *except* that you need to remember to say `$envelope = $envelope->with()` so that `$envelope` becomes the newly stamped object.

## Fetching Stamps

Also, when it comes to stamps, an `Envelope` could, in theory, hold *multiple* stamps of the same class. The `$envelope->last()` method says:

> *"Give me the most recently added `UniqueIdStamp` or null if there are none."*

## Dumping the Unique Id

Thanks to our work, below the if statement - regardless of whether this message was *just* dispatched... or just received from a transport... or is being retried - our `Envelope` should have exactly *one* `UniqueIdStamp`. Fetch it off with

`$stamp = $envelope->last(UniqueIdStamp::class)`. I'm also going to add a little hint to my editor so that it knows that this is specifically a `UniqueIdStamp`.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 8
class AuditMiddleware implements MiddlewareInterface
{
    public function handle(Envelope $envelope, StackInterface $stack): Envelope
    {
        if (null === $envelope->last(UniqueIdStamp::class)) {
            $envelope = $envelope->with(new UniqueIdStamp());
        }

        /** @var UniqueIdStamp $stamp */
        $stamp = $envelope->last(UniqueIdStamp::class);
// ... lines 19 - 21
    }
}
```

To see if this is working, let's `dump($stamp->getUniqueId())`.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 8
class AuditMiddleware implements MiddlewareInterface
{
    public function handle(Envelope $envelope, StackInterface $stack): Envelope
    {
        if (null === $envelope->last(UniqueIdStamp::class)) {
            $envelope = $envelope->with(new UniqueIdStamp());
        }

        /** @var UniqueIdStamp $stamp */
        $stamp = $envelope->last(UniqueIdStamp::class);
        dump($stamp->getUniqueId());

        return $stack->next()->handle($envelope, $stack);
    }
}
```

Let's try it! If we've done our job well, for an asynchronous message, that `dump()` will be executed once when the message is dispatched and *again* inside of the worker when it's received from the transport and handled.

Refresh the page just to be safe, then upload an image. To see if our `dump()` was hit, I'll use the link on the web debug toolbar to open up the profiler for that request. Click "Debug" on the left and... there it is! Our unique id! In a few minutes, we'll make sure that this code is *also* executed in the worker.

And because middleware are executed for *every* message, we should also be able to see this when *deleting* a message. Click that, then open up the profiler for the DELETE request and click "Debug". Ha! This time there are *two* distinct unique ids because deleting dispatches *two* different messages.

Next, let's actually do something useful with this! Inside of our middleware, we're going to log the *entire* lifecycle of a single message: when it's originally dispatched, when it's sent to a transport and when it's received from a transport and handled. To figure out which part of the process the message is currently in, we're going to once again use stamps. But instead of creating *new* stamps, we'll read the *core* stamps.

# Chapter 18: Logger Channel Setup and Autowiring

Here's our goal... and the end result is going to be pretty cool: leverage our middleware - *and* the fact that we're adding this unique id to every message - to log the entire lifecycle of a message to a file. I want to see when a message was originally dispatched, when it was sent to the transport, when it was received from the transport and when it was handled.

## Adding a Log Handler

Before we get into the middleware stuff, let's configure a new logger channel that logs to a new file. Open up `config/packages/dev/monolog.yaml` and add a new `channels` key. Wait... that's not right. A logging channel is, sort of a "category", and you can control *how* log messages for each category are handled. We don't want to add it *here* because then that new channel would *only* exist in the dev environment. Nope, we want the channel to exist in *all* environments... even if we decide to only give those messages special treatment in `dev`.

To do that, directly inside `config/packages`, create a new file called `monolog.yaml`... though... remember - the *names* of these config files aren't important. What *is* important is to add a `monolog` key, then `channels` set to an array with one new one - how about `messenger_audit`.

config/packages/monolog.yaml

```yaml
monolog:
    channels: [messenger_audit]
```

Thanks to this, we now have a new logger service in the container for this channel. Let's find it: at your terminal, run:

```
php bin/console debug:container messenger_audit
```

There it is: `monolog.logger.messenger_audit` - we'll use that in a minute. But first, I want to make any logs to this channel save to a new file in the `dev` environment. Back up in `config/packages/dev/monolog.yaml`, copy the `main` handler, paste and change the key

to `messenger`... though that could be anything. Update the file to be called `messenger.log` and - here's the magic - instead of saying: log all messages *except* those in the `event` channel, change this to *only* log messages that are *in* that `messenger_audit` channel.

config/packages/dev/monolog.yaml

```
monolog:
    handlers:
// ... lines 3 - 7
        messenger:
            type: stream
            path: "%kernel.logs_dir%/messenger.log"
            level: debug
            channels: ["messenger_audit"]
// ... lines 13 - 25
```

## Autowiring the Channel Logger

Cool! To use this service, we can't just autowire it by type-hinting the normal `LoggerInterface`... because that will give us the *main* logger. This is one of those cases where we have *multiple* services in the container that all use the same class or interface.

To make it wirable, back in `services.yaml`, add a new global bind: `$messengerAuditLogger` that points to the service id: copy that from the terminal, then paste as `@monolog.logger.messenger_audit`.

config/services.yaml

```
// ... lines 1 - 7
services:
// ... line 9
    _defaults:
// ... lines 11 - 12
        bind:
// ... lines 14 - 15
            $messengerAuditLogger: '@monolog.logger.messenger_audit'
// ... lines 17 - 34
```

Thank to this, if we use an argument named `$messengerAuditLogger` in the constructor of a service or in a controller, Symfony will pass us that service. By the way, starting in Symfony 4.2, instead of binding only to the *name* of the argument, you can also bind to the name *and* type by saying `Psr\Log\LoggerInterface $messengerAuditLogger`. That just makes things

more specific: Symfony would pass us this service for any arguments that have this name *and* the `LoggerInterface` type-hint.

Anyways, we have a new logger channel, that channel will log to a special file, and the logger service for that channel is wirable. Time to get to work!

Close up the monolog config files and go to `AuditMiddleware`. Add a `public function __construct()` with one argument `LoggerInterface $messengerAuditLogger` - the same name we used in the config. I'll call the property itself `$logger`, and finish this with `$this->logger = $messengerAuditLogger`.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 4
use Psr\Log\LoggerInterface;
// ... lines 6 - 10
class AuditMiddleware implements MiddlewareInterface
{
    private $logger;

    public function __construct(LoggerInterface $messengerAuditLogger)
    {
        $this->logger = $messengerAuditLogger;
    }
// ... lines 19 - 40
}
```

## Setting up the Context

Down in `handle()`, remove the `dump()` and create a new variable called `$context`. In addition to the actual log *message*, it's a little-known fact that you can pass extra information to the logger... which is *super* handy! Let's create a key called `id` set to the unique id, and another called `class` that's set to the class of the *original* message class. We can get that with `get_class($envelope->getMessage())`.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 10
11 class AuditMiddleware implements MiddlewareInterface
12 {
↕ // ... lines 13 - 19
20     public function handle(Envelope $envelope, StackInterface $stack): Envelope
21     {
↕ // ... lines 22 - 28
29         $context = [
30             'id' => $stamp->getUniqueId(),
31             'class' => get_class($envelope->getMessage())
32         ];
↕ // ... lines 33 - 39
40     }
41 }
```

Let's do the logging next! It's a bit more interesting than you might expect. How can we figure out if the current message was just *dispatched* or was just *received* asynchronously from a transport? And if it was just dispatched, how can we find out whether or not the message will be handled right now or sent to a transport for later? The answer... lies in the stamps!

# Chapter 19: Middleware Message Lifecycle Logging

Our middleware is called in *two* different situations. First, it's called when you initially dispatch the message. For example, in `ImagePostController`, the moment we call `$messageBus->dispatch()`, all the middleware are called - regardless of whether or not the message will be handled async. And second, when the worker - `bin/console messenger:consume` - receives a message from the transport, it passes that message *back* into the bus and the middleware are called again.

This is the trickiest thing about middleware: trying to figure out which situation you're currently in. Fortunately, Messenger adds "stamps" to the `Envelope` along the way, and *these* tell us *exactly* what's going on.

## Was the Message Received from the Transport? ReceivedStamp

For example, when a message is *received* from a transport, messenger adds a `ReceivedStamp`. So, if `$envelope->last(ReceivedStamp::class)`, then this message is currently being processed by the worker and was just received from a transport.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\Messenger\Stamp\ReceivedStamp;
10
11 class AuditMiddleware implements MiddlewareInterface
12 {
↕ // ... lines 13 - 19
20     public function handle(Envelope $envelope, StackInterface $stack):
   Envelope
21     {
↕ // ... lines 22 - 32
33         if ($envelope->last(ReceivedStamp::class)) {
↕ // ... line 34
35         } else {
↕ // ... line 36
37         }
↕ // ... lines 38 - 39
40     }
41 }
```

Let's log that: `$this->logger->info()` with a special syntax:

> *"[{id}] Received and handling {class}"*

Then pass `$context` as the second argument. The `$context` array is cool for two reasons. First, each log handler receives this and can do whatever it wants with it - usually the `$context` is printed at the end of the log message. And second, if you use these little `{}` wildcards, the context values will get filled in automatically!

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\Messenger\Stamp\ReceivedStamp;
10
11 class AuditMiddleware implements MiddlewareInterface
12 {
↕ // ... lines 13 - 19
20     public function handle(Envelope $envelope, StackInterface $stack): Envelope
21     {
↕ // ... lines 22 - 32
33         if ($envelope->last(ReceivedStamp::class)) {
34             $this->logger->info('[{id}] Received & handling {class}', $context);
35         } else {
↕ // ... line 36
37         }
↕ // ... lines 38 - 39
40     }
41 }
```

If the message was *not* just received, say `$this->logger->info()` and start the same way:

> *"[{id}] Handling or sending {class}"*

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\Messenger\Stamp\ReceivedStamp;
10
11 class AuditMiddleware implements MiddlewareInterface
12 {
↕ // ... lines 13 - 19
20     public function handle(Envelope $envelope, StackInterface $stack): Envelope
21     {
↕ // ... lines 22 - 32
33         if ($envelope->last(ReceivedStamp::class)) {
34             $this->logger->info('[{id}] Received & handling {class}', $context);
35         } else {
36             $this->logger->info('[{id}] Handling or sending {class}', $context);
37         }
↕ // ... lines 38 - 39
40     }
41 }
```

At this point, we know that the message was *just* dispatched... but we don't know whether or not it will be handled right now or sent to a transport. We'll improve that in a few minutes.

But first, let's try it! Start the worker and tell it to read from the `async` transport:

```
php bin/console messenger:consume -vv async
```

Ah, I think we had a few messages from earlier still in the queue! When that finishes, let's clear the screen. Let's also open up *another* tab and create the new log file - `messenger.log` - if it's not already there:

```
touch var/log/messenger.log
```

Then, tail it so we can watch the messages:

```
tail -f var/log/messenger.log
```

Oh, cool! This already has a few lines from those old messages it just processed. Let's clear that so we have fresh screens to look at.

Testing time! Move over and upload one new photo. Spin back to your terminal and... yea! Both log messages are already there: "Handling or sending" and then "Received and handling" when the message was received from the transport... which was almost instant. We know these log entries are for the *same* message thanks to the unique id at the beginning.

## Determining if Message is Handled or Sent

But... we can do better than just saying "handling *or* sending". How? This `$stack->next()->handle()` line is responsible for calling the *next* middleware... which will then call the *next* middleware and so on. Because our logging code is *above* this, it means that our code is potentially being called *before* some other middleware do their work. In fact, our

code is being executed before the core middleware that are responsible for handling or sending the message.

So... how can we determine whether the message will be sent versus handled immediately... before the message is *actually* sent or handled immediately? We can't!

Check it out: remove the `return` and instead say `$envelope = $stack->next()->handle()`. Then, move that line *above* our code and, at the bottom, `return $envelope`.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 11
12 class AuditMiddleware implements MiddlewareInterface
13 {
↕ // ... lines 14 - 20
21     public function handle(Envelope $envelope, StackInterface $stack): Envelope
22     {
↕ // ... lines 23 - 35
36         $envelope = $stack->next()->handle($envelope, $stack);
37
38         if ($envelope->last(ReceivedStamp::class)) {
39             $this->logger->info('[{id}] Received & handling {class}', $context);
40         } else {
41             $this->logger->info('[{id}] Handling or sending {class}', $context);
42         }
43
44         return $envelope;
45     }
46 }
```

If we did *nothing* else... the result would be pretty much the same: we would log the *exact* same messages... but technically, the log entries would happen *after* the message was sent or handled instead of before.

*But*! Notice that when we call `$stack->next()->handle()` to execute the rest of the middleware, we get back an `$envelope`... which *may* contain new stamps! In fact, *if* the message was sent to a transport instead of being handled immediately, it will be marked with a `SentStamp`.

Add `elseif $envelope->last(SentStamp::class)` then we know that this message was *sent*, *not* handled. Use `$this->logger->info()` with our `{id}` trick and `sent {class}`.

src/Messenger/AuditMiddleware.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Messenger\Stamp\SentStamp;
11
12 class AuditMiddleware implements MiddlewareInterface
13 {
↕ // ... lines 14 - 20
21     public function handle(Envelope $envelope, StackInterface $stack): Envelope
22     {
↕ // ... lines 23 - 37
38         if ($envelope->last(ReceivedStamp::class)) {
↕ // ... line 39
40         } elseif ($envelope->last(SentStamp::class)) {
41             $this->logger->info('[{id}] Sent {class}', $context);
42         } else {
↕ // ... line 43
44         }
↕ // ... lines 45 - 46
47     }
48 }
```

Below, now we know that we're definitely "Handling sync". The top message - "Received and handling" is still true, but I'll change this to just say "Received": a message is *always* handled when it's received, so that was redundant.

src/Messenger/AuditMiddleware.php

```
// ... lines 1 - 9
use Symfony\Component\Messenger\Stamp\SentStamp;

class AuditMiddleware implements MiddlewareInterface
{
// ... lines 14 - 20
    public function handle(Envelope $envelope, StackInterface $stack): Envelope
    {
// ... lines 23 - 37
        if ($envelope->last(ReceivedStamp::class)) {
            $this->logger->info('[{id}] Received {class}', $context);
        } elseif ($envelope->last(SentStamp::class)) {
            $this->logger->info('[{id}] Sent {class}', $context);
        } else {
            $this->logger->info('[{id}] Handling sync {class}', $context);
        }
// ... lines 45 - 46
    }
}
```

Ok! Let's clear our log screen and restart the worker:

```
php bin/console messenger:consume -vv async
```

Upload one photo... then move over... and go to the log file. Yep! Sent, then Received! If we had uploaded 5 photos, we could use the unique id to identify each message individually.

Hit enter a few times: I want to see an even *cooler* example. Delete a photo and move back over! Remember, this dispatches *two* messages! The unique id part makes it even *more* obvious what's going on: `DeletePhotoFile` was sent to the transport, then `DeleteImagePost` was handled synchronously... then `DeletePhotoFile` was received and processed.

Actually, what *really* happened was this: `DeleteImagePost` was handled synchronously and, internally, it dispatched `DeletePhotoFile` which was sent to the transport. The first two messages are a bit out of order because our logging code is always running *after* we execute the rest of the chain, so *after* `DeleteImagePost` was handled. We could improve that by moving the `Handling Sync` logging logic *above* the code that calls the rest of the middleware.

Yea, this stuff is *super* powerful... but can be a bit complex to navigate. This logging stuff is probably as confusing as it gets.

Next: the worker handles each message in the order it was received. But... that's not ideal: it's *way* more important for *all* `AddPonkaToImage` messages to be handled before *any* `DeletePhotoFile` messages. Let's do that with priority transports.

# Chapter 20: High Priority Transports

The two messages that we we're sending to the `async` transport are `AddPonkaToImage` and `DeletePhotoFile`, which handles deleting the physical file from the filesystem. And... that second one isn't something the user actually notices or cares about - it's just housekeeping. If it happened 5 minutes from now or 10 days from now, the user wouldn't care.

This creates an interesting situation. Our worker handles things in a first-in-first-out basis: if we send 5 messages to the transport, the worker will handle them in the order in which they were received. This means that if a *bunch* of images are deleted and *then* someone uploads a new photo... the worker will process *all* of those delete messages *before* finally adding Ponka to the photo. And that... isn't ideal.

The truth is that `AddPonkaToImage` messages should have a higher priority in our system than `DeletePhotoFile`: we *always* want `AddPonkaToImage` to be handled *before* any `DeletePhotoFile` messages... even if they were added first.

## Creating the "high" Priority Transport

So... can we set a priority on messages? Not exactly. It turns out that in the queueing world, this is solved by creating multiple *queues* and giving each of *those* a priority. In Symfony Messenger, that translates to multiple *transports*.

Below the `async` transport, create a new transport called, how about, `async_priority_high`. Let's use the same DSN as before, which in our case is using `doctrine`. Below, add `options`, then `queue_name` set to `high`. The name `high` isn't important - we could use anything. The `queue_name` option is specific to the Doctrine transport and ultimately controls the value of a column in the table, which operates like a category and allows us to have multiple "queues" of messages inside the same table. And also, for *any* transport, you can configure these options as query parameters on the DSN or under this `options` key.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 10
        transports:
// ... lines 12 - 17
            async_priority_high:
                dsn: '%env(MESSENGER_TRANSPORT_DSN)%'
                options:
                    queue_name: high
// ... lines 22 - 30
```

At this point we have three queues - which are all stored in the same table in the database, but with different `queue_name` values. And now that we have this new transport, we can route `AddPonkaToImage` to `async_priority_high`.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 25
        routing:
// ... line 27
            'App\Message\AddPonkaToImage': async_priority_high
// ... lines 29 - 30
```

## Consuming Prioritized Transports

If we stopped now... all we've *really* done is make it possible to send these two different message classes to two different queues. But there's nothing special about `async_priority_high`. Sure, I put the word "high" in its name, but it's no different than `async`.

The real magic comes from the worker. Find your terminal where the worker is running and hit Control+C to stop it. If you *just* run `messenger:consume` without any arguments and you have more than one transport, it asks you which transport you want to consume:

```
php bin/console messenger:consume
```

Meaning, which transport do you want to receive messages from. But actually, you can read messages from *multiple* transports at once *and* tell the worker which should be read first. Check this out: I'll say `async_priority_high, async`.

This tells the worker: first ask `async_priority_high` if it has any messages. If it doesn't, *then* go check the `async` transport.

We should be able to see this in action. I'll refresh the page, delete a *bunch* of images here as fast as I can and then upload a couple of photos. Check the terminal output:

It's handles `DeletePhotoFile` then... `AddPonkaToImage`, another `AddPonkaToImage`, *another* `AddPonkaToImage` and... yea! It goes back to handling the lower-priority `DeletePhotoFile`.

So, in the beginning - before we uploaded - it *did* consume a few `DeletePhotoFile` messages. But as soon as it saw a message on that `async_priority_high` transport, it consumed all of those until it was empty. When it was, it *then* returned to consuming messages from `async`.

Basically, each time the worker looks for the next message, it checks the highest priority transport first and *only* asks the next transport - or transports - if it's empty.

And... that's it! Create a new transport for however many different priority "levels" you need, then tell the worker command which order to process them. Oh, and instead of using this interactive way of doing things, you can run:

```
php bin/console messenger:consume async_priority_high async
```

Perfect. Next, let's talk about *one* option we can use to make it easier to develop while using queues... because *always* needing to remember to run the worker command while coding can be a pain.

# Chapter 21: Handling Messages Sync while Developing

I *love* the ability to defer work for later by sending messages to a transport. But, there *is* at least one practical bummer: it makes it a bit harder to actually *develop* and code your app. In addition to setting up your web server, database and anything else, you *now* need to remember to run:

```
php bin/console messenger:consume
```

Otherwse... things won't *fully* work. If you have a robust setup for local development - maybe something using Docker - you could build this right into that setup so that it runs automatically. Except... you'd *still* need to remember to *restart* the worker any time you make a change to some code that it uses.

It's not the *worst* thing ever. But, if this drives you crazy, there *is* a really nice solution: tell Messenger to handle all of your messages synchronously when you're in the `dev` environment.

## Hello "sync" Transport

Check out `config/packages/messenger.yaml`. One of the commented-out parts of this file is a, kind of, "suggested" transport called `sync`. The really important part isn't the name `sync` but the DSN: `sync://`. We learned earlier that Messenger supports *several* different *types* of transport like Doctrine, redis and AMQP. And the way you *choose* which one you want is the beginning of the connection string, like `doctrine://`. The `sync` transport is really neat: instead of *truly* sending each message to an external queue... it just handles them immediately. They're handled synchronously.

## Making the Transports sync

We can take advantage of this and use a configuration trick to change our `async` and `async_priority_high` transports to use the `sync://` transport *only* when we're in the `dev` environment.

Go into the `config/packages/dev` directory. Any files here are *only* loaded in the `dev` environment and *override* all values from the main `config/packages` directory. Create a new file called `messenger.yaml`... though the name of this file isn't important. Inside, we'll put the same configuration we have in our main file: `framework`, `messenger`, `transports`. Then override `async` and set it to `sync://`. Do the same for `async_priority_high`: set it to `sync://`.

config/packages/dev/messenger.yaml

```yaml
framework:
    messenger:
        transports:
            async: 'sync://'
            async_priority_high: 'sync://'
```

That's it! In the *dev* environment, *these* values will override the `dsn` values from the main file. And, we can see this: in an open terminal tab, run:

```
php bin/console debug:config framework messenger
```

This command shows you the real, *final* config under `framework` and `messenger`. And... yea! Because we're currently in the `dev` environment, both transports have a `dsn` set to `sync://`.

I *do* want to mention that the `queue_name` option is something that's specific to Doctrine. The `sync` transport doesn't use that, and so, it ignores it. It's possible that in a future version of Symfony, this would throw an error because we're using an undefined option for this transport. If that happens, we would just need to change the YAML format to set the `dsn` key - like we do in the main `messenger.yaml` file - and then override the `options` key and set it to an empty array. I'm mentioning that *just* in case.

Ok, let's try this! Refresh the page to be safe. Oh, and before we upload something, go back to the terminal where our worker is running, hit Control+C to stop it, and restart it. Woh! It's busted!

> *"You cannot receive messages from the sync transport."*

Messenger is saying:

> *"Yo! Um... the SyncTransport isn't a real queue you can read from... so stop trying to do it!"*

It's right... and this is exactly what we wanted: we wanted to be able to have our handlers called in the `dev` environment *without* needing to worry about running this command.

Ok, *now* let's try it: upload a couple of photos and... yea... it's *super* slow again. But Ponka *is* added when it finishes. The messages are being handled synchronously.

To make sure this is *only* happening for the `dev` environment, open up the `.env` file and change `APP_ENV` to be `prod` temporarily. Make sure to clear your cache so this works:

```
php bin/console cache:clear
```

Now, we *should* be able to run `messenger:consume` like before:

```
php bin/console messenger:consume -vv async_priority_high async
```

And... we can! Sync messages in dev, async in prod.

Now that we've accomplished this, change `APP_ENV` back to `dev` and, just to keep things more interesting for the tutorial, comment out the new `sync` config we just added: I want to continue using our *real* transports while we're coding. Stop and restart the worker:

config/packages/dev/messenger.yaml

```
framework:
    messenger:
#        transports:
#            async: 'sync://'
#            async_priority_high: 'sync://'
```

Now that we're back in the `dev` environment, stop and restart the worker:

```
php bin/console messenger:consume -vv async_priority_high async
```

Next: let's talk about a similar problem: how do you handle transports when writing automated tests?

# Chapter 22: Functional Test for the Upload Endpoint

How can we write automated tests for all of this? Well... I have so many answers for that. First, you could unit test your *message* classes. I don't *normally* do this... because those classes *tend* to be so simple... but if your class is a bit more complex or you want to play it safe, you can *totally* unit test this.

More important are the message handlers: it's *definitely* a good idea to test these. You could write unit tests and mock the dependencies or write an integration test... depending on what's most useful for what each handler does.

The point is: for message and message handler classes... testing them has absolutely *nothing* to do with messenger or transports or async or workers: they're just well-written PHP classes that we can test like *anything* else. That's really one of the beautiful things about messenger: above all else, you're just writing nice code.

But *functional* tests are more interesting. For example, open `src/Controller/ImagePostController.php`. The `create()` method is the upload endpoint and it does a couple of things: like saving the `ImagePost` to the database and, most important for us, dispatching the `AddPonkaToImage` object.

Writing a functional test for this endpoint is actually fairly straightforward. But what if we wanted to be able to test not *only* that this endpoint "appears" to have worked, but also that the `AddPonkaToImage` object *was*, in fact, sent to the transport? After all, we can't test that Ponka *was* actually added to the image because, by the time the response is returned, it hasn't happened yet!

## Test Setup

Let's get the functional test working first, before we get all fancy. Start by finding an open terminal and running:

```
composer require phpunit --dev
```

That installs Symfony's `test-pack`, which includes the PHPUnit bridge - a sort of "wrapper" around PHPUnit that makes life easier. When it finishes, it tells us to write our tests inside the `tests/` directory - brilliant idea - and execute them by running `php bin/phpunit`. That little file was just added by the recipe and it handles all the details of getting PHPUnit running.

Ok, step one: create the test class. Inside `tests`, create a new `Controller/` directory and then a new PHP Class: `ImagePostControllerTest`. Instead of making this extend the normal `TestCase` from PHPUnit, extend `WebTestCase`, which will give us the functional testing superpowers we deserve... and need. The class lives in FrameworkBundle but... be careful because there are (gasp) *two* classes with this name! The one you want lives in the `Test` namespace. The one you *don't* want lives in the `Tests` namespace... so it's super confusing. It should look like this. If you choose the wrong one, delete the `use` statement and try again.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 2
namespace App\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ImagePostControllerTest extends WebTestCase
{
// ... lines 9 - 12
}
```

*But*.... while writing this tutorial and getting mad about this confusing part, I created an issue on the Symfony repository. And I'm *thrilled* that by the time I recorded the audio, the other class has already been renamed! Thanks to janvt who jumped on that. Go open source!

Anyways, because we're going to test the `create()` endpoint, add `public function testCreate()`. Inside, to make sure things are working, I'll try my favorite `$this->assertEquals(42, 42)`.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 2
namespace App\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ImagePostControllerTest extends WebTestCase
{
    public function testCreate()
    {
        $this->assertEquals(42, 42);
    }
}
```

## Running the Test

Notice that I didn't get any auto-completion on this. That's because PHPUnit *itself* hasn't been downloaded yet. Check it out: find your terminal and run the tests with:

```
php bin/phpunit
```

This little script uses Composer to download PHPUnit into a separate directory in the background, which is nice because it means you can get any version of PHPUnit, even if some of its dependencies clash with those in your project.

Once it's done... ding! Our one test is green. And the next time we run:

```
php bin/phpunit
```

it jumps *straight* to the tests. And now that PHPUnit is downloaded, once PhpStorm builds its cache, that yellow background on `assertEquals()` will go away.

## Testing the Upload Endpoint

To test the endpoint itself, we *first* need an image that we can upload. Inside the `tests/` directory, let's create a `fixtures/` directory to hold that image. Now I'll copy one of the images I've been uploading into this directory and name it `ryan-fabien.jpg`.

There it is. The test itself is pretty simple: create a client with `$client = static::createClient()` and an `UploadedFile` object that will represent the file being uploaded: `$uploadedFile = new UploadedFile()` passing the path to the file as the first argument - `__DIR__.'/../fixtures/ryan-fabien.jpg` - and the filename as the second - `ryan-fabien.jpg`.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\File\UploadedFile;

class ImagePostControllerTest extends WebTestCase
{
    public function testCreate()
    {
        $client = static::createClient();

        $uploadedFile = new UploadedFile(
            __DIR__.'/../fixtures/ryan-fabien.jpg',
            'ryan-fabien.jpg'
        );
// ... lines 18 - 22
    }
}
```

Why the, sorta, "redundant" second argument? When you upload a file in a browser, your browser sends *two* pieces of information: the physical contents of the file *and* the name of the file on your filesystem.

Finally, we can make the request: `$client->request()`. The first argument is the method... which is `POST`, then the URL - `/api/images` - we don't need any GET or POST parameters, but we *do* need to pass an array of files.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\File\UploadedFile;

class ImagePostControllerTest extends WebTestCase
{
    public function testCreate()
    {
        $client = static::createClient();

        $uploadedFile = new UploadedFile(
            __DIR__.'/../fixtures/ryan-fabien.jpg',
            'ryan-fabien.jpg'
        );
        $client->request('POST', '/api/images', [], [
// ... line 19
        ]);
// ... lines 21 - 22
    }
}
```

If you look in `ImagePostController`, we're expecting the name of the uploaded file - that's normally the `name` attribute on the `<input` field - to literally be `file`. Not the *most* creative name ever... but sensible. Use that key in our test and set it to the `$uploadedFile` object.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\File\UploadedFile;

class ImagePostControllerTest extends WebTestCase
{
    public function testCreate()
    {
        $client = static::createClient();

        $uploadedFile = new UploadedFile(
            __DIR__.'/../fixtures/ryan-fabien.jpg',
            'ryan-fabien.jpg'
        );
        $client->request('POST', '/api/images', [], [
            'file' => $uploadedFile
        ]);

        dd($client->getResponse()->getContent());
    }
}
```

And... that's it! To see if it worked, let's just `dd($client->getResponse()->getContent())`.

tests/Controller/ImagePostControllerTest.php

```
↕ // ... lines 1 - 5
6  use Symfony\Component\HttpFoundation\File\UploadedFile;
7
8  class ImagePostControllerTest extends WebTestCase
9  {
10     public function testCreate()
11     {
12         $client = static::createClient();
13
14         $uploadedFile = new UploadedFile(
15             __DIR__.'/../fixtures/ryan-fabien.jpg',
16             'ryan-fabien.jpg'
17         );
18         $client->request('POST', '/api/images', [], [
19             'file' => $uploadedFile
20         ]);
21
22         dd($client->getResponse()->getContent());
23     }
24 }
```

Testing time! Find your terminal, clear the screen, deep breath and...

```
php bin/phpunit
```

Got it! And we get a new id each time we run it. The `ImagePost` records are saving to our *normal* database because I haven't gone to the trouble of creating a separate database for my `test` environment. That *is* something I normally like to do.

## Asserting Success

Remove the `dd()`: let's use a real assertion: `$this->assertResponseIsSuccessful()`.

tests/Controller/ImagePostControllerTest.php

```
↕ // ... lines 1 - 7
8  class ImagePostControllerTest extends WebTestCase
9  {
10     public function testCreate()
↕ // ... lines 11 - 21
22         $this->assertResponseIsSuccessful();
23     }
24 }
```

This nice method was added in Symfony 4.3... and it's not the only one: this new `WebTestAssertionsTrait` has a *ton* of nice new methods for testing a whole bunch of stuff!

If we stopped now... this is a nice test and you might be perfectly happy with it. But... there's one part that's *not* ideal. Right now, when we run our test, the `AddPonkaToImage` message is *actually* being sent to our transport... or at least we *think* it is... we're not actually verifying that this happened... though we can check manually right now.

To make this test more useful, we can do one of two different things. First, we could override the transports to be synchronous in the test environment - just like we did with `dev`. Then, if handling the message failed, our test would fail.

Or, second, we could *at least* write some code here that *proves* that the message was *at least* sent to the transport. Right now, it's possible that the endpoint could return 200... but some bug in our code caused the message never to be dispatched.

Let's add that check next, by leveraging a special "in memory" transport.

# Chapter 23: Testing with the "in-memory" Transport

A few minutes ago, in the `dev` environment only, we overrode all our transports so that all messages were handled synchronously. We commented it out for now, but this is *also* something that you could choose to do in your `test` environment, so that when you run the tests, the messages are handled *within* the test.

This may or may not be what you want. On one hand, it means your functional test is testing more. On the other hand, a functional test should probably test that the endpoint works and the message is sent to the transport, but testing the handler itself should be done in a test specifically for that class.

That's what we're going to do now: figure out a way to *not* run the handlers synchronously but *test* that the message *was* sent to the transport. Sure, if we killed the worker, we could query the `messenger_messages` table, but that's a bit hacky - and only works if you're using the Doctrine transport. Fortunately, there's a more interesting option.

Start by copying `config/packages/dev/messenger.yaml` and pasting that into `config/packages/test/`. This gives us messenger configuration that will *only* be used in the `test` environment. Uncomment the code, and replace `sync` with `in-memory`. Do that for both of the transports.

config/packages/test/messenger.yaml

```yaml
framework:
    messenger:
        transports:
            async: 'in-memory://'
            async_priority_high: 'in-memory://'
```

The `in-memory` transport is really cool. In fact, let's look at it! I'll hit `Shift+Shift` in PhpStorm and search for `InMemoryTransport` to find it.

This... is basically a fake transport. When a message is sent to it, it doesn't handle it or send it anywhere, it stores it in a property. If you were to use this in a real project, the messages would then disappear at the end of the request.

But, this is *super* useful for testing. Let's try it. A second ago, each time we ran our test, our worker *actually* started processing those messages... which makes sense: we really *were* delivering them to the transport. Now, I'll clear the screen and then run:

```
php bin/phpunit
```

It still works... but *now* the worker does nothing: the message isn't *really* being sent to the transport anymore and it's lost at the end of our tests. But! From within the test, we can now *fetch* that transport and *ask* it how many messages were sent to it!

## Fetching the Transport Service

Behind the scenes, every transport is actually a service in the container. Find your open terminal and run:

```
php bin/console debug:container async
```

There they are: `messenger.transport.async` and `messenger.transport.async_priority_high`. Copy the second service id.

We want to verify that the `AddPonkaToImage` message is sent to the transport, and we know that it's being routed to `async_priority_high`.

Back in the test, this is super cool: we can fetch the *exact* transport object that was just used from within the test by saying: `$transport = self::$container->get()` and then pasting the service id: `messenger.transport.async_priority_high`

tests/Controller/ImagePostControllerTest.php

```
↕ // ... lines 1 - 8
9  class ImagePostControllerTest extends WebTestCase
10 {
11     public function testCreate()
12     {
↕ // ... lines 13 - 25
26         $transport = self::$container->get('messenger.transport.async_priority_high');
↕ // ... line 27
28     }
29 }
```

This `self::$container` property holds the container that was actually used during the test request and is designed so that we can fetch *anything* we want out of it.

Let's see what this looks like: `dd($transport)`.

tests/Controller/ImagePostControllerTest.php

```
↕ // ... lines 1 - 8
9  class ImagePostControllerTest extends WebTestCase
10 {
11     public function testCreate()
12     {
↕ // ... lines 13 - 25
26         $transport = self::$container->get('messenger.transport.async_priority_high');
27         dd($transport);
28     }
29 }
```

Now jump back over to your terminal and run:

```
php bin/phpunit
```

Nice! This dumps the `InMemoryTransport` object and... the `sent` property *indeed* holds our *one* message object! All we need to do now is add an assertion for this.

Back in the test, I'm going to help out my editor by adding some inline docs to advertise that this is an `InMemoryTransport`. Below add `$this->assertCount()` to assert that we expect one message to be returned when we say `$transport->`... let's see... the method that you can call on a transport to get the sent, or "queued" messages is `get()`.

tests/Controller/ImagePostControllerTest.php

```
// ... lines 1 - 6
use Symfony\Component\Messenger\Transport\InMemoryTransport;

class ImagePostControllerTest extends WebTestCase
{
    public function testCreate()
    {
// ... lines 13 - 24
        /** @var InMemoryTransport $transport */
        $transport = self::$container->get('messenger.transport.async_priority_high');
        $this->assertCount(1, $transport->get());
    }
}
```

Let's try it! Run:

```
php bin/phpunit
```

Got it! We're now guaranteeing that the message was sent but we've kept our tests faster and more directed by not trying to handle them synchronously. If we were using something like RabbitMQ, we also don't need to have that running whenever we execute our tests.

Next, let's talk deployment! How do we run our workers on production... and make sure they stay running?

# Chapter 24: Deployment & Supervisor

So... how does all of this work on production? It's a simple problem really: on production, we somehow need to make sure that this command - `messenger:consume` - is *always* running. Like, *always*.

Some hosting platforms - like SymfonyCloud - allow you to do this with some simple configuration. You basically say:

> *"Yo Cloud provider thingy! Please make sure that `bin/console messenger:consume` is always running. If it quits for some reason, start a new one."*

If you're *not* using a hosting platform like that, it's ok - but you *will* need to do a little bit of work to get that same result. And actually, it's not *just* that we need a way to make sure that someone starts this command and then it runs forever. We actually *don't* want the command to run forever. No matter how well you write your PHP code, PHP just isn't meant to be ran *forever* - eventually your memory footprint will increase too much and the process will die. And... that's perfect! We *don't* want our process to run forever. Nope: what we *really* want is for `messenger:consume` to run, handle... a few messages... then close itself. Then, we'll use a *different* tool to make sure that each time the process disappears, it gets restarted.

## Hello Supervisor

The tool that does that is called supervisor. After you install it, you give it a command that you *always* want running and it stays up *all* night *constantly* eating pizza and watching to make sure that command is running. The *moment* it stops running, for *any* reason, it puts down the pizza and it restarts the command.

So let's see how Supervisor works and how we can use it to make sure our worker is *always* running. Because I'm using a Mac, I already installed Supervisor via Brew. If you're using Ubuntu, you can install it via apt. By the way, you don't *actually* need to install & configure Supervisor on your local machine: you only need it on production. We're installing it so we can test and make sure everything works.

## Supervisor Configuration

To get it going, we need a supervisor configuration file. Google for "Messenger Symfony" and open the main documentation. In the middle... there's a spot that talks about supervisor. Copy the configuration file. We could put this anywhere: it doesn't need to live in our project. But, I like to keep it in my repo so I can store it in git. In... how about `config/`, create a new file called `messenger-worker.ini` and paste the code inside.

config/messenger-worker.ini

```
[program:messenger-consume]
command=php /path/to/your/app/bin/console messenger:consume async --time-limit=3600
user=ubuntu
numprocs=2
autostart=true
autorestart=true
process_name=%(program_name)s_%(process_num)02d
```

The file tells Supervisor which command to run and other important info like which user it should run the process as and the *number* of processes to run. This will create *two* worker processes. The more workers you run, the more messages can be handled at once. But also, the more memory & CPU you'll need.

Now, locally, I don't need to run supervisor... because we can just manually run `messenger:consume`. But to make sure this all works, we're going to *pretend* like my computer is production and change the path to point to use my local path: `/Users/weaverryan/messenger`... which if I double-check in my terminal... oop - I forgot the `Sites/` part. Then, down here, I'll change the user to be `weaverryan`. Again, you would *normally* set this to your *production* values.

Oh, and if you look closely at the command, it's running `messenger:consume async`. Make sure to also consume `async_priority_high`. The command *also* has a `--time-limit=3600` option. We'll talk more about this and some other options in a bit, but this is great: it tells the worker to run for 60 minutes and then exit, to make sure it doesn't get too old and take up too much memory. As soon as it exits, Supervisor will restart it.

## Running Supervisor

Now that we have our config file, we need to make sure Supervisor can see it. Each Supervisor install has a *main* configuration file. On a Mac where it's installed via Brew, that file is located at `/usr/local/etc/supervisord.ini`. On Ubuntu, it should be `/etc/supervisor/supervisord.conf`.

Then, *somewhere* in your config file, you'll find an `include` section with a `files` line. *This* means that Supervisor is looking in this directory to find configuration files - like *ours* - that will tell it what to do.

To get *our* configuration file into that directory, we can create a symlink: `ln -s ~/Sites/messenger/config/messenger-worker.ini` then paste the directory.

```
ln -s ~/Sites/messenger/config/messenger-worker.ini /usr/local/etc/supervisor.d
```

Ok! Supervisor *should* now be able to see our config file. To *run* supervisor, we'll use something called `supervisorctl`. Because I'm on a Mac, I *also* need to pass a `-c` option and point to the configuration file we were just looking at. If you're on Ubuntu, you shouldn't need to do this - it'll know where to look already. Then say `reread`: that tells Supervisor to reread the config files:

```
supervisorctl -c /usr/local/etc/supervisord.ini reread
```

By the way, you *may* need to run this command with `sudo`. If you do, no big deal: it will execute the processes themselves as the user in your config file.

Cool! It sees the new `messager-consume` group. That names comes from the key at the top of our file. Next, run the `update` command... which would restart any processes with the new config... *if* they were already running... but our's aren't yet:

```
supervisorctl -c /usr/local/etc/supervisord.ini update
```

To start them, run `start messenger-consume:*`:

```
supervisorctl -c /usr/local/etc/supervisord.ini start messenger-consume:*
```

That last argument - `messenger-consume:*` isn't very obvious. When you create a "program" called `messenger-consume`, this creates what's called a "homogeneous process group". Because we have `processes=2`, this group will run *two* processes. By saying `messenger-consume:*` it tells Supervisor to start all processes inside that group.

When we run it... it doesn't say anything... but... our worker commands should now be running! Let's go stop our manual worker so that *only* the ones from Supervisor are running. Now,

```
tail -f var/log/messenger.log
```

This will make it really obvious whether or not our messages are being handled by those workers. Now, upload a few photos, delete a couple of items, move over and... yea! It's working! It's actually working almost twice as fast as normal because we have *twice* the workers.

And, *now* we can have some fun. First, we can see the process id's created by Supervisor by running:

```
ps -A | grep messenger:consume
```

**Tip**

You can also use `ps aux`, which will work on more operating systems.

There they are: 19915 and 19916. Let's kill one of those:

```
kill 19915
```

And run that again:

```
ps -A | grep messenger:consume
```

Yes! 19916 is still there but because we killed the other one, supervisor started a *new* process for it: 19995. Supervisor *rocks*.

Next, let's talk more about the options we can use to *purposely* make workers exit before they take up too much memory. We'll also talk about how to restart workers on deploy so that they see the new code *and* a little detail about how things can break if you update your message class.

# Chapter 25: Killing Workers Early & on Deploy

Run:

```
php bin/console messenger:consume --help
```

We saw earlier that this has an option called `--time-limit`, which you can use to tell the command to run for 60 minutes and then exit. The command *also* has two other options - `--memory-limit` - to tell the command to exit once its memory usage is above a certain level - or `--limit` - to tell it to run a specific *number* of messages and then exit. All of these are *great* options to use because we really *don't* want our `messenger:consume` command to run too long: we really just want it to handle a few messages, then exit. Restarting the worker is handled by Supervisor and doesn't take a huge amount of resources. All of these options cause the worker to exit *gracefully*, meaning, it only exits *after* a message has been fully handled, never in the *middle* of it. But, if you let your worker run too long and it runs out of memory... that *would* cause it to exit in the middle of handling a message and... well... that's not great. Use these options. You can even use *all* of them at once.

## Restarting Workers on Deploy

There's also a completely different situation when you want *all* of your workers to restart: whenever you deploy. We've seen *why* many times already: whenever we make a change to our code, we've been manually restarting the `messenger:consume` command so that the worker *sees* the new code. The same thing will happen on production: when you deploy, your workers *won't* see the new code until they exit and are restarted. Right now, that could take up to *six* minutes to happen! That is not okay. Nope, at the moment we deploy, we need all of or worker processes to exit, and we need that to happen gracefully.

Fortunately, Symfony has our back. Once again, run `ps -A` to see the worker processes.

```
ps -A | grep messenger:consume
```

Now, pretend we've just deployed. To stop all the workers, run:

```
php bin/console messenger:stop-workers
```

Check the processes again:

```
ps -A | grep messenger:consume
```

Ha! Perfect! The two new process ids *prove* that the workers were restarted! How does this work? Magic! I mean, *caching*. Seriously.

Behind the scenes, this command sends a signal to each worker that it should exit. But the workers are smart: they don't exit *immediately*, they finish whatever message they're handling and *then* exit: a graceful exit. To send this signal, Symfony actually sets a flag in the cache system - and each worker checks this flag. If you have a multi-server setup, you'll need to make sure that your Symfony "app cache" is stored in something like Redis or Memcache instead of the filesystem so that everyone can read those keys.

## What Happens when you Deploy Message Class Changes

There's *one* more detail you need to think about and it's due to the asynchronous nature of handling messages. Open up `AddPonkaToImage`. Imagine that our site is currently deployed and the `AddPonkaToImage` class looks like this. When someone uploads an image, we serialize this class and send it to the transport.

Imagine now that we have a bunch of these messages sitting in the queue at the moment we deploy a new version of our site. In this new version, we've refactored the `AddPonkaToImage` class: we've renamed `$imagePostId` to `$imagePost`. What will happen when those *old* versions of `AddPonkaToImage` are loaded from the queue?

The answer... the new `$imagePost` property will be null... and some non-existent `$imagePostId` property would be set instead. And that would probably cause your handler some serious trouble. So, *if* you need to tweak some properties on an existing message class, you have two options. First, don't: create a *new* message class instead. Then, *after* you deploy, remove the old message class. *Or* second, update the message class but, temporarily, keep both the old and new properties and make your handler smart enough to look for both. Again, after one deploy, or really, once you're sure all the old messages have been processed, you can remove the old stuff.

And... that's it! Use Supervisor to keep your processes running and the `messenger:stop-workers` command to restart on deploy. You are ready to put this stuff into production.

Before we keep going, I'm going to find my terminal and run:

```
supervisorctl -c /usr/local/etc/supervisord.ini stop messenger-consume:*
```

That stops the two processes. Now I'll run my worker manually:

```
php bin/console messenger:consume -vv async_priority_high async
```

This just makes life easier and more obvious locally: I can see the output from my worker.

Next: we've talked about commands & command handlers. Now it's time to talk about *events* and *event handlers*, how we can use Messenger as an event bus and... what the heck that means.

# Chapter 26: Events & Event Bus

Messenger is a "message bus". And it turns out that a "message" is a pretty generic term in computer science. In fact, there are *three* types of messages you'll commonly hear about.

## Messages: Commands, Events & Queries

The first type of message is a "command". And *that* is the type we've been creating so far: we create message classes that *sound* like a command: `AddPonkaToImage` or `DeleteImagePost` and whose handlers *do* some action. When you create message classes & handlers that look like this, you're using Messenger as a "command bus". And one of the, sort of, "rules" of commands is that each command should have exactly one handler. That's the "command" design pattern.

The *second* type of message is an "event". If you create an "event" class and pass it to Messenger, then you're using Messenger as an "event" bus. The difference between what a "command" class looks like and what an "event" class looks like is subtle: it comes down to naming conventions and what you're ultimately trying to accomplish. An event is dispatched *after* something happens and can have zero to many handlers. Don't worry, we'll see what this looks like soon.

The third type of message is a "query" and we'll talk about those later. For now, let's focus on understanding events and how they're different from commands... because... it *can* be super confusing. And Messenger, being a generic "message bus" works perfectly with either.

## Creating a Second Bus

Before we create our first event, I'll close a few things and then open `config/packages/messenger.yaml`. If our app leverages both commands *and* events, it's *totally* ok to use just *one* bus to handle all of that. But, in the interest of making our life a bit more difficult and learning more, let's continue to use our existing bus *only* as a command bus and create a *new* bus to only use with events.

To do that, under the `buses:` key, add a new one called, how about, `event.bus`. Set this to `~` which is null... just because we don't have any other configuration that we need to put here yet. *This* will cause a *new* `MessageBus` service to be added to the container.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
        buses:
// ... lines 4 - 7
            event.bus: ~
// ... lines 9 - 32
```

So far, whenever we needed the message bus - like in `ImagePostController` - we autowired it by using the `MessageBusInterface` type-hint. The question *now* is: how can we get access to the *new* message bus service?

Find your terminal and run:

```
php bin/console debug:autowiring
```

... which... explodes! My bad:

> *"Invalid configuration for path `framework.messenger`: you must specify `default_bus`"*

Copy the name of the default bus. Once you define more than one bus, you need a `default_bus` key set to your "main" bus. This tells Symfony which MessageBus service to pass us when we use the `MessageBusInterface` type-hint.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
        default_bus: messenger.bus.default

        buses:
// ... lines 6 - 9
            event.bus: ~
// ... lines 11 - 34
```

Try the `debug:autowiring` command again... and search for "mess".

```
php bin/console debug:autowiring
```

Ah, *now* we have *two* entries! This tells me that if we use the `MessageBusInterface` type-hint, we'll get the `messenger.bus.default` service. Ignore the `debug.traced` part - that's just Symfony adding some debug tools. But *now*, if you use the `MessageBusInterface` type-hint *and* you name the argument `$eventBus`, it will pass you the new event bus service!

This is a new feature in Symfony 4.2 where you can autowire things by a combination of the type-hint *and* argument name. Symfony took the name of our bus - `event.bus` - and made it possible to use the `$eventBus` argument name.

## Differences Between Buses

Great! We now know how to get the event bus! But.. what's the difference between these two buses? Do they behave differently? The answer is... no!

A bus is nothing more than a set of middleware. If you have two bus objects that have the same middleware... well then... those message buses effectively *are* identical! So, other than the fact that, so far, we've only added our `AuditMiddleware` to the first bus, these buses will work and act identically. That's why, even though I've created one service to handle commands and another service to handle events... ah... we really could send all our commands and events to just *one* service.

Next, let's create an event, learn what it looks like, why we might use them, and how they're different than commands.

# Chapter 27: Creating & Handling Events

So... what the heck *is* an event? Let me give you an example. Suppose a user registers on your site. When that happens, you do three things: save the user to the database, send them an email and add them to a CRM system. The code to do this might all live in a controller, a service *or* a `SaveRegisteredUserHandler` if you had a `SaveRegisteredUser` command.

This means that your service - or maybe your command handler - is doing *three* separate things. That's... not a huge deal. But if you need to suddenly do a *fourth* thing, you'll need to add even *more* code. Your service - or handler - violates the single responsibility principle that says that each function should only have to accomplish a *single* task.

This is *not* the end of the world - I often write code like this... and it doesn't usually bother me. But this code organization problem is *exactly* why events exist.

Here's the idea: if you have a command handler like `SaveRegisteredUser`, it's *supposed* to only perform its *principle* task: it should save the registered user to the database. If you follow this practice, it should *not* do "secondary" tasks, like emailing the user or setting them up in a CRM system. Instead, it should perform the main task and then dispatch an event, like `UserWasRegistered`. Then, we would have *two* handlers for that event: one that sends the email and one that sets up the user in the CRM. The command handler performs the main "action" and the event helps other parts of the system "react" to that action.

As far as Messenger is concerned, commands and events all look identical. The difference comes down to each supporting a different *design* pattern.

## The Secondary Task of DeleteImagePostHandler

And... *we* already have a situation like this! Look at `DeleteImagePost` and then `DeleteImagePostHandler`. The "main" job for this handler is to remove this `ImagePost` from the database. But it *also* has a second task: deleting the underlying file from the filesystem.

To do that, well, we're dispatching a *second* command - `DeletePhotoFile` - and *its* handler deletes the file. Guess what... this is the event pattern! Well, it's *almost* the event pattern. The

only difference is the *naming*: `DeletePhotoFile` sounds like a "command". Instead of "commanding" the system to do something, an event is more of an "announcement" that something *did* happen.

To fully understand this, let's back up and re-implement all of this fresh. Comment out the `$messageBus->dispatch()` call and then remove the `DeletePhotoFile` use statement on top.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 10
11 class DeleteImagePostHandler implements MessageHandlerInterface
12 {
↕ // ... lines 13 - 21
22     public function __invoke(DeleteImagePost $deleteImagePost)
23     {
↕ // ... lines 24 - 29
30         //$this->messageBus->dispatch(new DeletePhotoFile($filename));
31     }
32 }
```

Next, to get a clean start: remove the `DeletePhotoFile` command class itself and `DeletePhotoFileHandler`. Finally, in `config/packages/messenger.yaml`, we're routing the command we just deleted. Comment that out.

config/packages/messenger.yaml

```
1  framework:
2      messenger:
↕ // ... lines 3 - 29
30         routing:
↕ // ... lines 31 - 32
33             #'App\Message\DeletePhotoFile': async
```

Let's look at this with fresh eyes. We've successfully made `DeleteImagePostHandler` perform is *primary* job only: deleting the `ImagePost`. And now we're wondering: where should I put the code to do the *secondary* task of deleting the physical file? We could put that logic right here, or leverage an *event*.

## Creating the Event

Commands, events & their handlers look identical. In the `src/Message` directory, to start organizing things a bit better, let's create an `Event/` subdirectory. Inside, add a new class:

`ImagePostDeletedEvent`.

src/Message/Event/ImagePostDeletedEvent.php

```php
<?php

namespace App\Message\Event;

class ImagePostDeletedEvent
{
// ... lines 7 - 17
}
```

Notice the *name* of this class: that's *critical*. Everything so far has sounded like a command: we're running around our code base shouting: `AddPonkaToImage`! And `DeleteImagePost`! We sound bossy.

But with events, you're not using a strict command, you're notifying the system of something that just happened: we're going to fully delete the image post and *then* say:

> *"Hey! I just deleted an image post! If you care... uh... now is your chance to... uh... do something! But I don't care if you do or not."*

The event itself could be handled by... nobody... or it could have *multiple* handlers. Inside the class, we'll store any data we think might be handy. Add a constructor with a `string $filename` - knowing the filename of the deleted `ImagePost` might be useful. I'll hit Alt + Enter and go to "Initialize Fields" to create that property and set it. Then, at the bottom, I'll go to "Code -> Generate" - or Command + N on a Mac - and select "Getters" to generate this one getter.

src/Message/Event/ImagePostDeletedEvent.php

```php
<?php

namespace App\Message\Event;

class ImagePostDeletedEvent
{
    private $filename;

    public function __construct(string $filename)
    {
        $this->filename = $filename;
    }

    public function getFilename(): string
    {
        return $this->filename;
    }
}
```

You may have noticed that, other than its name, this "event" class looks *exactly* like the command we just deleted!

## Creating the Event Handler

Creating an event "handler" *also* looks identical to command handlers. In the `MessageHandler` directory, let's create another subdirectory called `Event/` for organization. Then add a new PHP class. Let's call this `RemoveFileWhenImagePostDeleted`. Oh... but make sure you spell that all correctly.

src/MessageHandler/Event/RemoveFileWhenImagePostDeleted.php

```php
<?php

namespace App\MessageHandler\Event;

// ... lines 5 - 6
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class RemoveFileWhenImagePostDeleted implements MessageHandlerInterface
{
// ... lines 11 - 21
}
```

This *also* follows a different naming convention. For commands, if a command was named `AddPonkaToImage`, we called the handler `AddPonkaToImageHandler`. The big difference between commands and events is that, while each command has exactly one handler - so using the "command name Handler" convention makes sense - each event could have *multiple* handlers.

But the inside of a handler looks the same: implement `MessageHandlerInterface` and then create our beloved `public function __invoke()` with the type-hint for the event class: `ImagePostDeletedEvent $event`.

src/MessageHandler/Event/RemoveFileWhenImagePostDeleted.php

```
↕ // ... lines 1 - 2
3  namespace App\MessageHandler\Event;
4
5  use App\Message\Event\ImagePostDeletedEvent;
↕ // ... line 6
7  use Symfony\Component\Messenger\Handler\MessageHandlerInterface;
8
9  class RemoveFileWhenImagePostDeleted implements MessageHandlerInterface
10 {
↕ // ... lines 11 - 17
18     public function __invoke(ImagePostDeletedEvent $event)
19     {
↕ // ... line 20
21     }
22 }
```

Now... we'll do the work... and this will be identical to the handler we just deleted. Add a constructor with the one service we need to delete files: `PhotoFileManager`. I'll initialize fields to create that property then, down below, finish things with `$this->photoFileManager->deleteImage()` passing that `$event->getFilename()`.

src/MessageHandler/Event/RemoveFileWhenImagePostDeleted.php

```php
// ... lines 1 - 2
namespace App\MessageHandler\Event;

use App\Message\Event\ImagePostDeletedEvent;
use App\Photo\PhotoFileManager;
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class RemoveFileWhenImagePostDeleted implements MessageHandlerInterface
{
    private $photoFileManager;

    public function __construct(PhotoFileManager $photoFileManager)
    {
        $this->photoFileManager = $photoFileManager;
    }

    public function __invoke(ImagePostDeletedEvent $event)
    {
        $this->photoFileManager->deleteImage($event->getFilename());
    }
}
```

I hope this was *delightfully* boring for you. We deleted a command and command handler... and replaced them with an event and an event handler that are... other than the name... identical!

Next, let's dispatch this new event... but to our *event* bus. Then, we'll tweak that bus a little bit to make sure it works perfectly.

# Chapter 28: Dispatching the Event & No Handlers

Back in `DeleteImagePostHandler`, we need to dispatch our new `ImagePostDeletedEvent` message. Earlier, we created a *second* message bus service. We now have a bus that we're using as a command bus called `messenger.bus.default` and another one called `event.bus`. Thanks to this, when we run:

```
php bin/console debug:autowiring mess
```

we can now autowire *either* of these services. *Just* using the `MessageBusInterface` type-hint will give us the main command bus. But using that type-hint *plus* naming the argument `$eventBus` will give us the other.

Inside `DeleteImagePostHandler`, change the argument to `$eventBus`. I don't have to, but I'm also going to rename the property to `$eventBus` for clarity. Oh, and variables need a `$` in PHP. Perfect!

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 11
12 class DeleteImagePostHandler implements MessageHandlerInterface
13 {
14     private $eventBus;
↕ // ... lines 15 - 16
17     public function __construct(MessageBusInterface $eventBus,
   EntityManagerInterface $entityManager)
18     {
19         $this->eventBus = $eventBus;
↕ // ... line 20
21     }
↕ // ... lines 22 - 32
33 }
```

Inside `__invoke()`, it's really the same as before: `$this->eventBus->dispatch()` with `new ImagePostDeletedEvent()` passing that `$filename`.

src/MessageHandler/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 5
6 use App\Message\Event\ImagePostDeletedEvent;
↕ // ... lines 7 - 11
12 class DeleteImagePostHandler implements MessageHandlerInterface
13 {
↕ // ... lines 14 - 22
23     public function __invoke(DeleteImagePost $deleteImagePost)
24     {
↕ // ... lines 25 - 30
31         $this->eventBus->dispatch(new ImagePostDeletedEvent($filename));
32     }
33 }
```

That's it! The end result of all of this work... was to do the *same* thing as before, but with some renaming to match the "event bus" pattern. The handler performs its *primary* task - deleting the record from the database - then dispatches an event that says:

> *"An image post was just deleted! If anyone cares... do something!"*

## Routing Events

In fact, unlike with commands, when we dispatch an event... we don't actually care if there are *any* handlers for it. There could be zero, 5, 10 - we don't care! We're not going to use any return values from the handlers and, unlike with commands, we're not going to *expect* that *anything* specific happened. You're just screaming out into space:

> *"Hey! An ImagePost was deleted!"*

Anyways, the last piece we need to fix to make this *truly* identical to before is, in `config/packages/messenger.yaml`, down under `routing`, route `App\Message\Event\ImagePostDeletedEvent` to the `async` transport.

config/packages/messenger.yaml

```
1 framework:
2     messenger:
↕ // ... lines 3 - 29
30         routing:
↕ // ... lines 31 - 32
33             'App\Message\Event\ImagePostDeletedEvent': async
```

Let's try this! Find your worker and restart it. All of this refactoring was around deleting images so... let's delete a couple of things, move back over and... yea! It's working great! `ImagePostDeletedEvent` is being dispatched and handled.

## Handle Some Handlers Async?

Oh, and side note about routing. When you route a command class, you know *exactly* which *one* handler it has. And so, it's super easy to think about what that handler does and determine whether or not it can be handled async.

With events, it's a bit more complicated: this *one* event class could have *multiple* handlers. And, in theory, you might want some to be handled immediately and *others* later. Because Messenger is built around routing the *messages* to transports - not the handlers - making some handlers sync and others async isn't natural. However, if you need to do this, it *is* possible: you can route a message to *multiple* transports, then configure Messenger to only call *one* handler when it's received from transport A and only the *other* handler when it's received from transport B. It's a bit more complex, so I don't recommend doing this unless you need to. We won't talk about *how* in this tutorial, but it's in the docs.

## Events can have No Handlers

Anyways, I mentioned before that, for events, it's legal on a philosophical level to have *no* handlers... though you probably won't do that in your application because... what's the point of dispatching an event with no handlers? But... for the sake of trying it, open `RemoveFileWhenImagePostDeleted` and take off the `implements MessageHandleInterface` part.

src/MessageHandler/Event/RemoveFileWhenImagePostDeleted.php

```
↕ // ... lines 1 - 8
9  class RemoveFileWhenImagePostDeleted
10 {
↕ // ... lines 11 - 21
22 }
```

I'm doing this temporarily to see what happens if Symfony sees *zero* handlers for an event. Let's... find out! Back in the browser, try to delete an image. It works! Wait... oh, I forgot to stop

the worker... let's do that... then try again. This time... it works... but in the worker log... CRITICAL error!

> *"Exception occurred while handling `ImagePostDeletedEvent`: no handler for message."*

By default, Messenger *requires* each message to have at *least* one handler. That's to help us avoid silly mistakes. But... for an event bus... we *do* want to allow *zero* handlers. Again... this is more of a philosophical problem than a real one: it's unlikely you'll decide to dispatch events that have no handlers. But, let's see how to fix it!

In `messenger.yaml`, take the `~` off of `event.bus` and add a new option below: `default_middleware: allow_no_handlers`. The `default_middleware` option defaults to `true` and its *main* purpose is to allow you to set it to `false` if, for some reason, you wanted to *completely* remove the default middleware - the middleware that handle & send the messages, among other things. But you can also set it to `allow_no_handlers` if you want to *keep* the normal middleware, but *hint* to the `HandleMessageMiddleware` that it should *not* panic if there are zero handlers.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 4
        buses:
// ... lines 6 - 9
            event.bus:
                default_middleware: allow_no_handlers
// ... lines 12 - 35
```

Go back and restart the worker. Then, delete another image... come back here and... cool! It says "No handler for message" but it doesn't freak out and cause a failure.

So now our command bus and event bus *do* have a small difference... though they're still *almost* identical... and we could *really* still get away with sending both commands and events through the same bus. Put the `MessageHandlerInterface` back on the class... and restart our worker one more time.

src/MessageHandler/Event/RemoveFileWhenImagePostDeleted.php

```
↕ // ... lines 1 - 8
9  class RemoveFileWhenImagePostDeleted implements MessageHandlerInterface
10 {
↕ // ... lines 11 - 21
22 }
```

Now that we're feeling good about events... I have a question: what's the difference between dispatching an event into Messenger versus dispatching an event into Symfony's EventDispatcher?

Let's talk about that next.

# Chapter 29: Messenger vs EventDispatcher

If you've ever create an event listener or event subscriber in Symfony, you're creating a "listener" for an event that's dispatched through a service called the "event dispatcher". The purpose of the event dispatcher is to allow one piece of code to "notify" the app that something happened and for anyone else to "listen" to that event and run some code.

Which... huh... is the *exact* same purpose of dispatching an event into Messenger. What the heck? If I want to dispatch an event in my code, should I use the EventDispatcher or Messenger? Are animated image files pronounced "jif" or "gif"? Should toilet paper hang "over" the roll or "under"? Ahh!

## Messenger can be Async

First, there *is* a practical difference between dispatching an event to the EventDispatcher versus Messenger: Messenger allows your handlers to be called *asynchronously*, whereas listeners to events from the EventDispatcher are *always* synchronous.

## EventDispatcher communicates back

And this leads to a nice rule of thumb. Whenever you dispatch an event, *if* you want listeners to that event to be able to communicate *back* to you, so you can then do something based on their feedback, use the EventDispatcher. But if you simply want to say "this thing happened" and you don't need any feedback from possible listeners or handlers, use Messenger.

For example, in `AddPonkaToImageHandler`, suppose we wanted to dispatch an event here so that other parts of the system could tell us exactly *which* Ponka image should be added to this photo. In that case, we need those listeners to be able to communicate *back* to us. To do that we would create an Event class that holds the `ImagePost` object *and* has a setter on it that listeners can call - maybe `setPonkaImageToUse()`. We would then use the `EventDispatcher` and dispatch the message *before* actually adding Ponka to the image. Once all the listeners were called, we could see if any of them called that `setPonkaImageToUse()` method.

But what if we simply wanted to say:

> *“Hey! We just added Ponka to an image!”*

... and didn't need any information back from possible handlers? In that case we would create a similar event class, leave *off* the `setPonkaImageToUse()` method and dispatch it with Messenger. Messenger is perfect if you don't need any info back from your handlers because... those handlers might end up being called asynchronously!

If it's *still* not clear to you, just use whichever you want. Why? Because if you end up wanting your code to run asynchronously, you'll end up choosing Messenger. And if you want your listeners to be able to communicate back to the code that dispatches the messages, you'll use EventDispatcher. Otherwise, either will work.

Next, let's use some service configuration tricks to tighten up how we've organized our commands, command handlers, events and event handlers.

# Chapter 30: Doctrine Transaction & Validation Middleware

We're now using both a command bus pattern, where we create commands and command handlers, and the event bus pattern: we have our first event and event handler. The difference between a command and event... is a little subtle. Each command should have exactly one handler: we're *commanding* that something perform a specific action: `AddPonkaToImage`. But an event is something that's usually dispatched *after* that action is taken, and the purpose is to allow anyone else to take any *secondary* action - to *react* to the action.

## Two Buses... Why?

Obviously, Messenger itself is a generic enough tool that it can be used for both of these use cases. Open up `config/packages/messenger.yaml`. We decided to register one bus service that we're using as our *command* bus and a *separate* bus service that we're using as our event bus. But... there's really almost *no* difference between these two buses! A bus is nothing more than a collection of middleware... so the *only* differences are that the first has `AuditMiddleware`... which we could also add to the second... and we told the `HandleMessageMiddleware` on the event bus to allow "no handlers" for a message: if an event has *zero* handlers, it won't throw an exception.

But really... this is *so* minor that if you wanted to use just *one* bus for everything, that would work great.

## Validation, Doctrine Transaction, etc Middleware

However, there *are* some people that make their command and event buses a bit *more* different. Google for "Symfony Messenger multiple buses" to find an article that talks about how to manage multiple buses. In this example, the docs show *three* different buses: the command bus, a query bus - which we'll talk about in a minute - and an event bus. But each bus has *slightly* different middleware.

These two middleware - `validation` and `doctrine_transaction` - come automatically with Symfony but aren't enabled by default. If you add the `validation` middleware, when you dispatch a message, that middleware will *validate* the message object *itself* through Symfony's validator. If validation fails, it will throw a `ValidationFailedException` that you can catch in your code to read off the errors.

This is cool... but we're *not* using this because I prefer to validate my data *before* sending it into the bus. It just makes more sense to me and looks a bit simpler than a, somewhat, "invisible" layer doing validation for us. But, it's a totally valid thing to use.

The `doctrine_transaction` middleware is similar. If you activate this middleware, it will wrap your handler inside a Doctrine transaction. If the handler throws an exception, it will rollback the transaction. And if *no* exception is thrown, it will commit it. This means that your handler won't need to call `flush()` on the EntityManager: the middleware does that for you.

This is cool... but I'm ok with creating and managing Doctrine transactions myself if I need them. So, this is another nice middleware that I *like*, but don't use.

Anyways, if you *do* use more middleware than we're using, then your different buses *might* start to... actually be *more* different... and using multiple bus services would make more sense. Like with everything, if the simpler approach - using one bus for everything - is working for you, great! Do that. If you need flexibility to have different middleware on different buses, awesome. Configure multiple buses.

Since *multiple* buses is the more complex use-case... and we're deep-diving into Messenger, let's keep our multiple bus setup and get our code organized even better around this concept.

## Messages Sent to Wrong Bus

Find your terminal and run:

```
php bin/console debug:messenger
```

Ah... Now that we have multiple buses, it breaks down the information on a bus-by-bus basis. It says that the following messages can be dispatched to our command bus and... huh... these *same* messages are allowed to be dispatched to the event bus.

That's... ok... but it's not *really* want we want. *We* know that certain messages are *commands* and will be sent to the command bus and others are events. But when we set up our handlers, we never told Messenger that *this* handler should only be used by *this* bus. So, Messenger makes sure that *all* buses are aware of *all* handlers. That's not a huge deal, but it means that if we accidentally took this command and dispatched it to the event bus, it would work! And if we took this event and sent it to the command bus, *it* would work. If we're relying on each bus to have quite different middleware, we probably *don't* want to make that mistake.

So... we're going to do something *totally* optional... but nice, when you're using events and commands. Look inside the `Message` and `MessageHandler` directories: we have a mixture of events and commands. Sure, I put the event into an `Event/` subdirectory, but we haven't done the same for commands.

Let's do that next: let's organize our message & message handlers better. Once we do this, we can use a service configuration trick to make sure that the command bus *only* knows about the command handlers and the event bus *only* knows about the event handlers.

# Chapter 31: Event & Command Bus Organization

We already organized our new event class into an `Event` subdirectory. Cool! Let's do the same thing for our commands. Create a new `Command/` sub-directory, move the two command classes inside... then add `\Command` to the end of the namespace on both classes.

Let's see... now that we've changed those namespaces... we need to update a few things. Start in `messenger.yaml`: we're referencing `AddPonkaToImage`. Add `Command` to that class name. Next, in `ImagePostController`, all the way on top, we're referencing *both* commands. Update the namespace on each one.

And finally, in the handlers, we have the same thing: each handler has a `use` statements for the command class it handles. Add the `Command\` namespace on both.

Cool! Let's do the same thing for the handlers: create a new subdirectory called `Command/`, move those inside... then add the `\Command` namespace to each one. That's... all we need to change.

I like it! There was nothing technical about this change... it's just a nice way to organize things if you're planning to use more than just commands - meaning events or query messages. And everything will work exactly the same way it did before. To prove it, at your terminal, run `debug:messenger`:

```
php bin/console debug:messenger
```

Yep! We see the same info as earlier.

## Binding Handlers to One Bus

But... now that we've separated our event handlers from our command handlers... we can do something special: we can *tie* each handler to the *specific* bus that it's intended for. Again, it's not *super* important to do this, but it'll tighten things up.

Let me show you: open up `config/services.yaml`. This `App\` line is responsible for auto-registering every class in the `src/` directory as a service in the container.

The line below *repeats* that for classes in the `Controller/` directory. Why? This will *override* the controller services registered above and add a special *tag* that controllers need to work.

We can use a similar trick with Messenger. Say `App\MessageHandler\Command\`, then use the `resource` key to re-auto-register all the classes in the `../src/MessageHandler/Command` directory. Whoops - I typo'ed that directory name - I'll see a *huge* error in a few minutes... and will fix that.

config/services.yaml

```
↕ // ... lines 1 - 7
8 services:
↕ // ... lines 9 - 29
30     App\MessageHandler\Command\:
31         resource: '../src/MessageHandler/Command'
↕ // ... lines 32 - 44
```

If we *only* did this... absolutely *nothing* would change. This would register everything in this directory as a service... but that's already done by the first `App\` entry anyways.

But *now* we can add a tag to this with `name: messenger.message_handler` and `bus:` set to... the name of my bus from `messenger.yaml`. Copy `messenger.bus.default` and say `bus: messenger.bus.default`.

config/services.yaml

```
↕ // ... lines 1 - 7
8 services:
↕ // ... lines 9 - 29
30     App\MessageHandler\Command\:
31         resource: '../src/MessageHandler/Command'
↕ // ... line 32
33         tags: [{ name: messenger.message_handler, bus:
   messenger.bus.default }]
↕ // ... lines 34 - 44
```

There are a few things going on here. First, when Symfony sees a class in our code that implements `MessageHandlerInterface`, it *automatically* adds this `messenger.message_handler` tag. *This* is how Messenger knows which classes are message *handlers*.

We're now adding that tag *manually* so that we can *also* say exactly which *one* bus this handler should be used on. Without the `bus` option, it's added to *all* buses.

We also need to add one more key: `autoconfigure: false`.

config/services.yaml

```
// ... lines 1 - 7
services:
// ... lines 9 - 29
    App\MessageHandler\Command\:
        resource: '../src/MessageHandler/Command'
        autoconfigure: false
        tags: [{ name: messenger.message_handler, bus:
messenger.bus.default }]
// ... lines 34 - 44
```

Thanks to the `_defaults` section on top, all services in our `src/` directory will, by default, have `autoconfigure` *enabled*... which is the feature that's responsible for automatically adding the `messenger.message_handler` tag to all services that implement `MessageHandlerInterface`. We're turning it *off* for services in this directory so that the tag isn't added *twice*.

Phew! You can see the end result by running `debug:messenger` again.

```
php bin/console debug:messenger
```

Oh, the end result is a huge error thanks to my typo! Make sure you're referencing the `MessageHandler` directory. Try `debug:messenger` again:

```
php bin/console debug:messenger
```

Nice! The event bus *no longer* says that we can dispatch the two commands two it. What this *really* means is that the command handlers were added to the command bus, but *not* to the event bus.

Let's repeat this for the events: copy this section, paste, change the namespace to `Event\`, the directory to `Event` and update the `bus` option to `event.bus` - the name of our other bus

inside `messenger.yaml`.

config/services.yaml

```
// ... lines 1 - 7
services:
// ... lines 9 - 34
    App\MessageHandler\Event\:
        resource: '../src/MessageHandler/Event'
        autoconfigure: false
        tags: [{ name: messenger.message_handler, bus: event.bus }]
// ... lines 39 - 44
```

Cool! Try `debug:messenger` again:

```
php bin/console debug:messenger
```

Perfect! Our two command handlers are bound to the command bus and our one event handler is tied to the event bus.

Again, doing this last step wasn't *that* important... but I *do* really like these sub-directories... and tightening things up is nice.

## Renaming the Command Bus

Oh, but while we're cleaning things up, back in `config/packages/messenger.yaml`, our main bus is called `messenger.bus.default`, which becomes the bus's service id in the container. We used this name... just because that's the default value Symfony uses when you have only *one* bus. But because this is a *command* bus, let's... call it that! Rename it to `command.bus`. And above, use that as our `default_bus`.

config/packages/messenger.yaml

```
framework:
    messenger:
        default_bus: command.bus

        buses:
            command.bus:
// ... lines 7 - 35
```

Where was the old key referenced in our code? Thanks to the fact that we autowire that service via its type-hint... almost nowhere - just in `services.yaml`. Change the bus option to `command.bus` as well.

config/services.yaml

```
// ... lines 1 - 7
services:
// ... lines 9 - 29
    App\MessageHandler\Command\:
// ... lines 31 - 32
        tags: [{ name: messenger.message_handler, bus: command.bus }]
// ... lines 34 - 44
```

Check everything out by running `debug:messenger` one more time:

```
php bin/console debug:messenger
```

That's nice: two buses, each with a great name and only aware of the correct handlers.

Oh, and this `AuditMiddleware` is something that we really should also use on `event.bus`: it logs the journey of messages... which is equally valid here.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 4
        buses:
// ... lines 6 - 9
            event.bus:
// ... line 11
                middleware:
                    - App\Messenger\AuditMiddleware
// ... lines 14 - 37
```

If you love this organization, great! If it seems like too much, keep it simple. Messenger is here to do what you want. Next, let's talk about the last type of message bus: the query bus.

# Chapter 32: Query Bus

The last type of bus that you'll hear about is... the double-decker tourist bus! I mean... the query bus! Full disclosure... while I *am* a fan of waving like an idiot on the top-level of a tourist bus, I'm *not* a huge fan of query buses: I think they make your code a bit more complex... for not much benefit. That being said, I want you to *at least* understand what it is and how it fits into the message bus methodology.

## Creating the Query Bus

In `config/packages/messenger.yaml` we have `command.bus` and `event.bus`. Let's add `query.bus`. I'll keep things simple and just set this to `~` to get the default settings.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 4
        buses:
// ... lines 6 - 14
            query.bus: ~
// ... lines 16 - 39
```

## What is a Query?

Ok: so what *is* the point of a "query bus"? We understand the purpose of commands: we dispatch messages that *sound* like commands: `AddPonkaToImage` or `DeleteImagePost`. Each command then has exactly one handler that performs that work... but doesn't *return* anything. I haven't really mentioned that yet: commands *just* do work, but they *don't* communicate anything *back*. Because of this, it's ok to process commands synchronously or asynchronously - our code isn't waiting to get information back from the handler.

A query bus is the *opposite*. Instead of commanding the bus to do work, the point of a *query* is to get information back *from* the handler. For example, let's pretend that, on our homepage, we

want to print the number of photos that have been uploaded. This is a *question* or *query* that we want to ask our system:

> *"How many photos are in the database?"*

If you're using the query bus pattern, instead of getting that info directly, you'll dispatch a *query*.

## Creating the Query & Handler

Inside the `Message/` directory, create a new `Query/` subdirectory. And inside of that, create a new PHP class called `GetTotalImageCount`.

Even that *name* sounds like a query instead of a command: I want to get the total image count. And... in this case, we can leave the query class blank: we won't need to pass any extra data to the handler.

src/Message/Query/GetTotalImageCount.php

```php
<?php

namespace App\Message\Query;

class GetTotalImageCount
{

}
```

Next, inside of `MessageHandler/`, do the same thing: add a `Query/` subdirectory and then a new class called `GetTotalImageCountHandler`. And like with *everything* else, make this implement `MessageHandlerInterface` and create `public function __invoke()` with an argument type-hinted with the message class: `GetTotalImageCount $getTotalImageCount`.

src/MessageHandler/Query/GetTotalImageCountHandler.php

```php
<?php

namespace App\MessageHandler\Query;

use App\Message\Query\GetTotalImageCount;
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class GetTotalImageCountHandler implements MessageHandlerInterface
{
    public function __invoke(GetTotalImageCount $getTotalImageCount)
    {
// ... line 12
    }
}
```

What do we do inside of here? Find the image count! Probably by injecting the `ImagePostRepository`, executing a query and then *returning* that value. I'll leave the querying part to you and just `return 50`.

src/MessageHandler/Query/GetTotalImageCountHandler.php

```php
<?php

namespace App\MessageHandler\Query;

use App\Message\Query\GetTotalImageCount;
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class GetTotalImageCountHandler implements MessageHandlerInterface
{
    public function __invoke(GetTotalImageCount $getTotalImageCount)
    {
        return 50;
    }
}
```

But hold on a second... cause we just did something *totally* new! We're returning a value from our handler! This is *not* something that we've done *anywhere* else. Commands do work but *don't* return any value. A query doesn't really do any work, its *only* point is to return a value.

Before we dispatch the query, open up `config/services.yaml` so we can do our same trick of binding each handler to the correct bus. Copy the `Event\` section, paste, change `Event` to `Query` in both places... then set the bus to `query.bus`.

config/services.yaml

```
// ... lines 1 - 7
services:
// ... lines 9 - 39
    App\MessageHandler\Query\:
        resource: '../src/MessageHandler/Query'
        autoconfigure: false
        tags: [{ name: messenger.message_handler, bus: query.bus }]
// ... lines 44 - 49
```

Love it! Let's check our work by running:

```
php bin/console debug:messenger
```

Yep! `query.bus` has one handler, `event.bus` has one handler and `command.bus` has two.

## Dispatching the Message

Let's do this! Open up `src/Controller/MainController.php`. This renders the homepage and so *this* is where we need to know how many photos have been uploaded. To get the query bus, we need to know which type-hint & argument name combination to use. We get that info from running:

```
php bin/console debug:autowiring mess
```

We can get the main `command.bus` by using the `MessageBusInterface` type-hint with *any* argument name. To get the query bus, we need to use that type-hint *and* name the argument: `$queryBus`.

Do that: `MessageBusInterface $queryBus`. Inside the function, say `$envelope = $queryBus->dispatch(new GetTotalImageCount())`.

src/Controller/MainController.php

```
↕ // ... lines 1 - 6
7  use Symfony\Component\Messenger\MessageBusInterface;
↕ // ... lines 8 - 9
10 class MainController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(MessageBusInterface $queryBus)
16     {
17         $envelope = $queryBus->dispatch(new GetTotalImageCount());
↕ // ... lines 18 - 19
20     }
21 }
```

## Fetching the Returned Value

We haven't used it too much, but the `dispatch()` method *returns* the final Envelope object, which will have a number of different stamps on it. One of the properties of a *query* bus is that every query will *always* be handled synchronously. Why? Simple: we need the answer to our query... *right* now! And so, our handler must be run *immediately*. In Messenger, there's nothing that *enforces* this on a query bus... it's just that we won't ever route our queries to a transport, so they'll always be handled right now.

Anyways, once a message is handled, Messenger automatically adds a stamp called `HandledStamp`. Let's get that: `$handled = $envelope->last()` with `HandledStamp::class`. I'll add some inline documentation above that to tell my editor that this will be a `HandledStamp` instance.

src/Controller/MainController.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\Messenger\Stamp\HandledStamp;
↕ // ... lines 9 - 10
11 class MainController extends AbstractController
12 {
↕ // ... lines 13 - 15
16     public function homepage(MessageBusInterface $queryBus)
17     {
↕ // ... lines 18 - 19
20         /** @var HandledStamp $handled */
21         $handled = $envelope->last(HandledStamp::class);
↕ // ... lines 22 - 26
27     }
28 }
```

So... why did we get this stamp? Well, we need to know what the *return* value of our handler was. And, conveniently, Messenger stores that on this stamp! Get it with `$imageCount = $handled->getResult()`.

src/Controller/MainController.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\Messenger\Stamp\HandledStamp;
↕ // ... lines 9 - 10
11 class MainController extends AbstractController
12 {
↕ // ... lines 13 - 15
16     public function homepage(MessageBusInterface $queryBus)
17     {
↕ // ... lines 18 - 19
20         /** @var HandledStamp $handled */
21         $handled = $envelope->last(HandledStamp::class);
22         $imageCount = $handled->getResult();
↕ // ... lines 23 - 26
27     }
28 }
```

Let's pass that into the template as an `imageCount` variable....

src/Controller/MainController.php

```
// ... lines 1 - 7
use Symfony\Component\Messenger\Stamp\HandledStamp;
// ... lines 9 - 10
class MainController extends AbstractController
{
// ... lines 13 - 15
    public function homepage(MessageBusInterface $queryBus)
    {
// ... lines 18 - 19
        /** @var HandledStamp $handled */
        $handled = $envelope->last(HandledStamp::class);
        $imageCount = $handled->getResult();

        return $this->render('main/homepage.html.twig', [
            'imageCount' => $imageCount
        ]);
    }
}
```

and then in the template - `templates/main/homepage.html.twig` - because our entire frontend is built in Vue.js, let's override the `title` block on the page and use it there: `Ponka'd {{ imageCount }} Photos`.

templates/main/homepage.html.twig

```
// ... lines 1 - 2
{% block title %}Ponka'd {{ imageCount }} Photos{% endblock %}
// ... lines 4 - 10
```

Let's check it out! Move over, refresh and... it works! We've Ponka's 50 photos... at least according to our hardcoded logic.

So... that's a query bus! It's not my favorite because we're not guaranteed what *type* it returns - the `imageCount` could really be a string... or an object of *any* class. Because we're not calling a *direct* method, the data we get back feels a little fuzzy. Plus, because queries need to be handled synchronously, you're not saving any performance by leveraging a query bus: it's purely a programming pattern.

But, my opinion is *totally* subjective, and a lot of people love query buses. In fact, we've been talking mostly about the *tools* themselves: command, event & query buses. But there are some deeper patterns like CQRS or event sourcing that these tools can unlock. This is not something we currently use here on SymfonyCasts... but if you're interested, you can read more about this topic - Matthias Noback's blog is my favorite source.

Oh, and before I forget, if you look back on the Symfony docs... back on the main messenger page... all the way at the bottom... there's a spot here about getting results from your handler. It shows some shortcuts that you can use to more easily get the value from the bus.

Next, let's talk about message handler *subscribers*: an alternate way to configure a message handler that has a few extra options.

# Chapter 33: Advanced Handler Config: Handler Subscribers

Open up `DeleteImagePostHandler`. The *main* thing that a message bus needs to know is the *link* between the `DeleteImagePost` message class and its handler. It needs to know that when we dispatch a `DeleteImagePost` object, it should call `DeleteImagePostHandler`.

How does Messenger know these two classes are connected? It knows because our handler implements `MessageHandlerInterface` - this "marks" it as a message handler - *and* because its `__invoke()` method is *type-hinted* with `DeleteImagePost`. If you follow these two rules - implement that interface & create an `__invoke()` method with an argument type-hinted with the message class - then... you're done!

Find your terminal and run:

```
php bin/console debug:messenger
```

Yep! This proves it: `DeleteImagePost` is handled by `DeleteImagePostHandler`.

Then... in `config/services.yaml`, we got a little bit fancier. By organizing each *type* of message - commands, events and queries - into different directories, we were able to add a *tag* to each service. This gives a bit *more* info to Messenger. It says:

> *"Hey! I want you to make that normal connection between the `DeleteImagePost` message class and `DeleteImagePostHandler`... but I only want you to tell the "command bus" about that connection... because that's the only bus I'm going to dispatch that message into."*

We also see this on `debug:messenger`: the command bus is aware of the `DeleteImagePost` and `DeleteImagePostHandler` connection and the other two buses know about *other* message and message handler links. Oh, and as a reminder, if this whole "tags" thing confuses you... skip it. It organizes things a bit more, but you can just as effectively have *one* bus that handles everything.

Anyways, this system is quick to use but there are a *few* things that you *can't* change. For example, the method in your handler *must* be called `__invoke()`... that's just what Symfony looks for. And because a class can only have one method named `__invoke()`, this means that you can't have a single handler that handles multiple different message classes. I don't *usually* like to do this anyways, I prefer one message class per handler... but it *is* a technical limitation.

## MessageHandlerInterface

Now that we've reviewed *all* of that... it turns out that this is only *part* of the story. If we want to, we can take *more* control of how a message class is linked to its handler... including some extra config.

How? Instead of implementing `MessageHandlerInterface`, implement `MessageSubscriberInterface`.

src/MessageHandler/Command/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Messenger\Handler\MessageSubscriberInterface;
↕ // ... lines 11 - 12
13 class DeleteImagePostHandler implements MessageSubscriberInterface
14 {
↕ // ... lines 15 - 38
39 }
```

This is less of a huge change than it may seem. If you open up `MessageSubscriberInterface`, it *extends* `MessageHandlerInterface`. So, we're still *effectively* implementing the same interface... but now we're forced to have one new method: `getHandledMessages()`.

At the bottom of my class, I'll go to Code -> Generate - or Command + N on a Mac - and select "Implement Methods".

As soon as we implement this interface, instead of magically looking for the `__invoke()` method and checking the type-hint on the argument for which message class this should handle, Symfony will call this method. Our job here? Tell it *exactly* which classes we handle, which method to call and... some *other* fun stuff!

src/MessageHandler/Command/DeleteImagePostHandler.php

```
// ... lines 1 - 12
class DeleteImagePostHandler implements MessageSubscriberInterface
{
// ... lines 15 - 34
    public static function getHandledMessages(): iterable
    {
        // TODO: Implement getHandledMessages() method.
    }
}
```

## Message Handling Config

The easiest thing you can put here is `yield DeleteImagePost::class`. Don't over-think that yield... it's just syntax sugar. You could also return an array with a `DeleteImagePost::class` string inside.

src/MessageHandler/Command/DeleteImagePostHandler.php

```
// ... lines 1 - 12
class DeleteImagePostHandler implements MessageSubscriberInterface
{
// ... lines 15 - 34
    public static function getHandledMessages(): iterable
    {
        yield DeleteImagePost::class;
    }
}
```

What difference did that make? Go back and run `debug:messenger`.

```
php bin/console debug:messenger
```

And... it made absolutely *no* difference. With this *super* simple config, we've told Messenger that this class handles `DeleteImagePost` objects... and then Messenger *still* assumes that it should execute a method called `__invoke()`.

But technically, this type-hint isn't needed anymore. Delete that, then re-run:

```
php bin/console debug:messenger
```

It *still* sees the connection between the message class and handler.

## Controlling the Method & Handling Multiple Classes

Ok... but since we probably *should* use type-hints... this isn't that interesting yet. What else can we do?

Well, by assigning this to an array, we can add some config. For example, we can say `'method' => '__invoke'`. Yep, we can now *control* which method Messenger will call. That's especially useful if you decide that you want to add *another* yield to handle a *second* message... and want Messenger to call a *different* method.

src/MessageHandler/Command/DeleteImagePostHandler.php

```
// ... lines 1 - 12
class DeleteImagePostHandler implements MessageSubscriberInterface
{
// ... lines 15 - 34
    public static function getHandledMessages(): iterable
    {
        yield DeleteImagePost::class => [
            'method' => '__invoke'
        ];
    }
}
```

## Handler Priority

What else can we put here? One option is `priority` - let's set it to... 10.

src/MessageHandler/Command/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 12
13 class DeleteImagePostHandler implements MessageSubscriberInterface
14 {
↕ // ... lines 15 - 34
35     public static function getHandledMessages(): iterable
36     {
37         yield DeleteImagePost::class => [
38             'method' => '__invoke',
↕ // ... lines 39 - 41
42             'priority' => 10,
43         ];
44     }
45 }
```

This option is... much less interesting than it might look like at first. We talked earlier about priority transports: in `config/packages/messenger.yaml` we created *two* transport - `async` & `async_priority_high` - and we route different messages to each. We did this so that, when we run our worker, we can tell it to always read messages from `async_priority_high` first before reading messages from `async`. That makes `async_priority_high` a place for us to send "higher" priority messages.

The `priority` option here is... less powerful. If you send a message to a transport with a priority 0 and then you send *another* message to that *same* transport with priority 10, what do you think will happen? Which message will be handled first?

The answer: the first message that was sent - the one with the *lower* priority. Basically, Messenger will *always* read messages in a first-in-first-out basis: it will *always* read the *oldest* messages first. The `priority` does *not* influence this.

So... what does it do? Well, if `DeleteImagePost` had *two* handlers... and one had the default priority of zero and another had 10, the handler with priority 10 would be called first. That's not *usually* important, but could be if you had two event handlers and *really* needed them to happen in a certain order.

Next, let's talk about *one* more option you can pass here - the most *powerful* option. It's called `from_transport` and allows you to, sort of, send different "handlers" of a message to different transports so that each can be consumed independently.

# Chapter 34: Sending Handlers to Different Transports: from_transport

The last option I want to mention *is* interesting... but can also be confusing. It's called `from_transport`.

If you look at `messenger.yaml`, this `DeleteImagePost` isn't being routed anywhere, which means it's handled synchronously. Let's pretend that we want to handle it *asynchronously*... and that we're routing it to the `async` transport. Set `from_transport` to `async`...

src/MessageHandler/Command/DeleteImagePostHandler.php

```
↕ // ... lines 1 - 12
13 class DeleteImagePostHandler implements MessageSubscriberInterface
14 {
↕ // ... lines 15 - 34
35     public static function getHandledMessages(): iterable
36     {
37         yield DeleteImagePost::class => [
↕ // ... lines 38 - 44
45             'from_transport' => 'async'
46         ];
47     }
48 }
```

then temporarily route this class to that transport in `messenger.yaml`.

Now, pretend that the `DeleteImagePost` message actually has *two* handlers... something that's very possible for events. Assuming that we did *not* add this `from_transport` config yet, if you sent `DeleteImagePost` to the `async` transport, then when that message is read from that transport by a worker, *both* handlers will be executed one after another.

But what if you wanted to, sort of, send *one* handler of that message to *one* transport, maybe `async_priority_high`, and *another* handler to *another* transport. Well, in Messenger, you don't send "handlers"... you send messages... and when Messenger consumes a message, it calls *all* the handlers for that message. Does that mean it's impossible to make one handler of a message "high" priority and another one low? Nope! This workflow *is* possible.

## Route to Two Transports

First, route `DeleteImagePost` to *both* the `async` and `async_priority_high` transports.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 34
        routing:
// ... lines 36 - 38
            'App\Message\Command\DeleteImagePost': [async, async_priority_high]
```

If we *only* did this, the message would be sent to *both* transports, it would be consumed *two* times, and *every* handler would be called *twice*... which is totally *not* what we want... unless each handler is baking cookies... or something.

But when we add this `from_transport` option set to `async`, it means that this handler should *only* be called when a `DeleteImagePost` object is consumed *from* the `async` transport. If we configured a *second* handler with `from_transport` set to `async_priority_high`, that handler would *only* be called when the message is being consumed from *that* transport.

In other words, you're sending the message to *two* transports, but each transport knows that it should only execute *one* handler. This allows your two handlers to be queued and executed by workers independently of each other. It's a *really* powerful feature... but because Messenger is centered around sending *messages* to transports, over-using this *can* be confusing.

Let's comment that out and remove the routing config.

src/MessageHandler/Command/DeleteImagePostHandler.php

```php
// ... lines 1 - 12
class DeleteImagePostHandler implements MessageSubscriberInterface
{
// ... lines 15 - 34
    public static function getHandledMessages(): iterable
    {
        yield DeleteImagePost::class => [
// ... lines 38 - 44
            //'from_transport' => 'async'
        ];
    }
}
```

That's basically it for the options you can pass here... though you can always check `MessageSubscriberInterface`: it talks about what's available.

Next, let's up our queueing game by changing from the Doctrine transport to RabbitMQ - also commonly referred to as AMQP. It's buckets of fun!

# Chapter 35: AMQP with RabbitMQ

Open up your `.env` file and check out the `MESSENGER_TRANSPORT_DSN` setting. We've been using the doctrine transport type. The `doctrine://default` string says that messages should be stored using Doctrine's `default` connection. In `config/packages/messenger.yaml`, we're referencing this environment variable for both the `async` and `async_priority_high` transports.

So... yep! We've been storing messages in a database table. It was quick to set up, easy to use - because we already understand databases - and robust enough for most use-cases.

## Hello AMQP... RabbitMQ

But the *industry* standard "queueing system" or "message broker" is *not* a database table, it's something called AMQP, or "Advanced Message Queuing Protocol". AMQP is... not *itself* a technology... it's a "standard" for how a, so-called, "message broker system" should work. Then, different queuing systems can "implement" this standard. Honestly, *usually* when someone talks about AMQP, they're talking about one specific tool: RabbitMQ.

Here's the idea: in the same way that you launch a "database server" and make queries to it, you can launch a "Rabbit MQ instance" then send messages to it and receive messages from it. On a high level... it doesn't work much differently than our simple database table: you put messages in... then ask for them later.

So... what *are* the advantages of using RabbitMQ instead of Doctrine? Maybe... nothing! What I mean is, if you *just* use the standard Messenger features and never dig deeper, both will work just fine. But if you have a highly-scaled system or want to use some advanced, RabbitMQ-specific features, well... then... RabbitMQ is the answer!

What are those more advanced features? Well, stick with me over the next few chapters and you'll start to uncover them.

## Launching an Instance via CloudAMQP.com

The easiest way to spin up a RabbitMQ instance is via `cloudamqp.com`: an awesome service for cloud-based RabbitMQ... with a free tier so we can play around! After logging in, create a new instance, give it a name, select any region... yep we *do* want the free tier and... "Create instance".

## AMQP Transport Configuration

Cool! Click into the new instance to find... a beautiful AMQP connection string! Copy that, go find our `.env` file... and paste over `doctrine://default`. You can also put this into a `.env.local` file... which is what I would *normally* do so I can avoid committing these credentials.

> **Tip**
>
> The URL that you copied will now start with `amqps://` (with an "s"!). That is "secure" AMQP. Change it to `amqp://` to get things working. Support for SSL was introduced in Symfony 5.2, but requires extra configuration.

Anyways, the `amqp://` part activates the AMQP transport in Symfony... and the rest of this contains a username, password and other connection details. As *soon* as we make this change, both our `async` and `async_priority_high` transports... are now using RabbitMQ! That was easy!

Oh, but notice that I *am* still using `doctrine` for my *failure* transport... and I'm going to keep that. The failure transport is a special type of transport... and it turns out that the `doctrine` transport *type* actually has the *most* features for reviewing failed messages. You *can* use AMQP for this, but I recommend Doctrine.

Before we try this, I want to make *one* other change. Open up `src/Controller/ImagePostController.php` and find the `create()` method. This is the controller that's executed whenever we upload a photo... and it's responsible for dispatching the `AddPonkaToImage` command. It *also* adds a 500 millisecond delay via this stamp. Comment that out for now... I'll show you *why* we're doing this a bit later.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             //new DelayStamp(500)
66         ]);
↕ // ... lines 67 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

## The AMQP PHP Extension

Ok! Other than removing that delay, *all* we've done is swap our transport config from Doctrine to AMQP. Let's... see if things still work! First, make sure your worker is *not* running... to begin with. Then, find your browser, select a photo and... it worked! Well, hold on... because you *may* have gotten a *big* AJAX error. If you did, open the profiler for that request. I'm *pretty* sure I know what error you'll see:

> *"Attempted to load class "AMQPConnection" from the global namespace. Did you forget a "use" statement?"*

Why... no we did not! Under the hood, Symfony's AMQP transport type uses a PHP *extension* called... well... amqp! It's an add-on to PHP - like xdebug or pdo_mysql - that you'll *probably* need to install.

The *pain* with PHP extensions is that installing them can vary based on your system. For Ubuntu, you may be able to run

```
sudo apt-get install php-amqp
```

Or you might use pecl, like I did with my Homebrew Mac install:

```
pecl install amqp
```

Once you *do* manage to get it installed, make sure to restart the Symfony web server so that it sees the change. If you're having issues getting this configured, let us know in the comments and we'll do our best to help!

When it *is* all configured, you should be able upload a photo with *no* errors. And... because this had no errors... it... *probably* just got sent to RabbitMQ? When I refresh, it says "Ponka is napping"... because nothing has *consumed* our message yet. Well, let's see what happens. Find your terminal and consume messages from both of our transports:

```
php bin/console messenger:consume -vv async_priority_high async
```

And... there it is! It received the message, handled it... and it's done! When we refresh the page... there's Ponka! It worked! Switching from Doctrine to RabbitMQ was as simple as changing our connection string.

Next, let's dig deeper into what just happened behind the scenes: what does it mean to "send" a message to RabbitMQ or "get" a message from it? Oh, and you're going to *love* the RabbitMQ debugging tools.

# Chapter 36: AMQP Internals: Exchanges & Queues

We've just changed our messenger configuration to send messages to a cloud-based RabbitMQ instance instead of sending them to Doctrine to be stored in the database. And after we made that change... everything... just kept working! We can send messages like normal and consume them with the `messenger:consume` command. That's awesome!

But I want to look a bit more at how this *works*... what's *actually* happening inside of RabbitMQ. Stop the worker... and then lets go delete a few images: one, two, three. This should have caused *three* new messages to be sent to Rabbit.

When we were using the Doctrine transport, we could query a database table to see these. Can we do something similar with RabbitMQ? Yea... we can! RabbitMQ comes with a *lovely* tool called the RabbitMQ Manager. Click to jump into it.

Aw yea, we've got data! And if we learn what some of these terms mean... this data will even start to make sense!

## Exchanges

The *first* big concept in RabbitMQ is an *exchange*... and, for me, this was the most confusing part of learning how Rabbit works. When you send a message to RabbitMQ, you send it to a specific *exchange*. Most of these exchanges were automatically created for us... and you can ignore them. But see that `messages` exchange? That was created by *our* application and, right now, *all* messages that Messenger transports to RabbitMQ are being sent to *this* exchange.

You won't see the name of this exchange in our messenger config yet, but each transport that uses AMQP has an `exchange` option and it *defaults* to `messages`. See this "Type" column? Our exchange is a type called `fanout`. Click into this exchange to get more info... and open up "bindings". This exchange has a "binding" to a "queue" that's... by coincidence... *also* called "messages".

## Exchanges Send to Queues

And *this* is where things can get a little confusing... but it's *really* a simple idea. The two main concepts in RabbitMQ are *exchanges* and *queues*. We're a lot more familiar with the idea of a queue. When we used the Doctrine transport type, our database table was basically a queue: it was a big list of queued messages... and when we ran the worker, it read messages from that list.

In RabbitMQ, queues have the same role: queues hold messages and we *read* messages from queues. So then... what the heck do these *exchange* things do?

The *key* difference between the Doctrine transport type and AMQP is that with AMQP you do *not* send a message directly to a queue. You can't say:

> *"Hey RabbitMQ! I would like to send this message to the `important_stuff` queue."*

Nope, in RabbitMQ, you send messages to an *exchange*. Then, that exchange will have some config that *routes* that message to a specific queue... or possibly multiple queues. The "Bindings" represents that config.

The *simplest* type of exchange is this `fanout` type. It says that each message that's sent to this exchange should be sent to *all* the queues that have been bound to it... which in our case is just one. The "binding" rules can get a lot smarter - sending different messages to different queues - but let's worry about that later. For now, this *whole* fancy setup means that *every* message will ultimately end up in a queue called `messages`.

Let's click on the Queues link on top. Yep, we have exactly *one* queue: `messages`. And... hey! It has *3* messages "Ready" inside of it, waiting for us to consume them!

## auto_setup Exchange & Queues

By the way... who *created* the `messages` exchange and `messages` queue? Are they... just standard to RabbitMQ? Rabbit *does* come with some exchanges out-of-the-box, but *these* were created by *our* app. Yep, like with the Doctrine transport-type, Messenger's AMQP transport has an `auto_setup` option that defaults to true. This means that it will detect if the exchange and queue it needs exist, and if they're don't, it will automatically create them. Yep, Messenger took care of creating the exchange, creating the queue *and* tying them together with the exchange

binding. Both the exchange name *and* queue name are options that you can configure on your transport... and both default to the word `messages`. We'll see that config a bit later.

## Send to an Exchange, Read from a Queue

To summarize *all* of this: we *send* a message to an exchange and it forwards it to one or more queues based on some internal rules. Whoever is "sending" - or "producing" - the message just says:

> *“Go to the exchange called "messages"”*

... and in theory... the "sender" doesn't really know or care what queue that message will end up in. Once the message *is* in a queue... it just sits there.. and waits!

The second part of the equation is your "worker" - the thing that *consumes* messages. The worker is the *opposite* of the sender: it doesn't know *anything* about *exchanges*. It just says:

> *“Hey! Give me the next message in the "messages" queue.”*

We send messages to exchanges, RabbitMQ routes those to queues, and we consume from those queues. The exchange is a new, extra layer... but the end-result is still pretty simple.

Phew! Before we try to run our worker, let's upload 4 photos. Then.... if you look at the `messages` queue... and refresh.... there it is! It has 7 messages!

## Consuming from the Queue

As a reminder, we're sending `AddPonkaToImage` messages to `async_priority_high` and `ImagePostDeletedEvent` to `async`. The idea is that we can put different messages into different queues and then consume messages in the `async_priority_high` queue before consuming messages in the `async` queue. The problem is that... right now... everything is ending up in the *same*, *one* queue!

Check this out - find your terminal and *only* consume from the `async` transport. This *should* cause *only* the `ImagePostDeletedEvent` messages to be consumed:

```
php bin/console messenger:consume -vv async
```

And... yup, it does handle a few `ImagePostDeletedEvent` objects. But if you keep watching... once it finishes those, it *does* start processing the `AddPonkaToImage` messages.

We have *such* a simple AMQP setup right now that we've introduced a bug: our two transports are *actually* sending to the exact same queue... which kills our ability to consume them in a prioritized way. We'll fix that next by using *two* exchanges.

Oh, but if you flip back over to the RabbitMQ manager - you can see all the messages being consumed. Cool stuff.

# Chapter 37: AMQP Priority Exchange

The idea behind our `async` and `async_priority_high` transports was that we can send some messages to `async_priority_high` and others to `async`, with the *goal* that those messages would end up in different "buckets"... or, more technically, in different "queues". Then we can instruct our worker to *first* read all messages from whatever queue `async_priority_high` is bound to *before* reading messages from whatever queue the `async` transport is bound to.

## The queue_name Option in Doctrine

This *did* work before with Doctrine, thanks to this `queue_name: high` option. The default value for this option is... `default`. As a reminder, I'll quickly log into my database:

```
mysql -u root messenger_tutorial
```

And see what that table looked like:

```
DESCRIBE messenger_messages;
```

Yep, the `queue_name` column was the key to making this work. Messages that were sent to `async_priority_high` had a `queue_name` set to `high`, and those sent to the `async` transport had a value of `default`. So even though we only had one database table, it functioned like two queues: when we consumed the `async_priority_high` transport, it queried for all messages `WHERE queue_name="high"`.

The *problem* is that this `queue_name` option is specific to the *doctrine* transport, and it has absolutely *no* effect when using AMQP.

## Routing Messages to... a Queue?

But... on a high-level... our goal is the same: we need *two* queues. We need the `async_priority_high` transport to send messages to *one* queue and the `async` transport to send messages to a *different* queue.

But with AMQP... you don't send a message directly to a queue... you send it to an *exchange*... and then it's the exchange's responsibility to look at its internal rules and figure out which queue, or queues, that message should actually go to.

This means that to get a message to a queue, we need to tweak things on the *exchange* level. And there are *two* different ways to do this. First, we could continue to have a *single* exchange and then add some internal rules - called *bindings* - to teach the exchange that *some* messages should go to one queue and *other* messages should go to *another* queue. I'm going to show you how to do this a bit later.

The second option isn't quite as cool, but it's a bit simpler. By default, when Messenger creates an exchange, it creates it as a `fanout` type. That means that when a message is sent to this exchange, it's routed to *every* queue that's bound to it. So if we added a *second* binding to a second queue - maybe `messages_high_priority` - then *every* message that's sent to this exchange would be routed to *both* queues. It would be duplicated! That's... not what we want.

Instead, we're going to create *two* `fanout` exchanges, and each exchange will route all of its messages to a *separate* queue. We'll have two exchanges and two queues.

## Configuring a Second Exchange

Let's configure this inside of `messenger.yaml`. Under `options` add `exchange` then `name` set to, how about, `messages_high_priority`. Below this, add `queues` with just one key below: `messages_high` set to `null`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... lines 21 - 26
            async_priority_high:
// ... line 28
                options:
                    exchange:
                        name: messages_high_priority
                    queues:
                        messages_high: ~
// ... lines 34 - 42
```

This config has *three* effects. First, because we have the `auto_setup` feature enabled, the first time we talk to RabbitMQ, Messenger will create the `messages_high_priority` exchange, the `messages_high` queue *and* bind them together. The *second* effect is that when we *send* messages to this transport they will be sent to the `messages_high_priority` exchange. The third and *final* effect is that when we *consume* from this transport, Messenger will read messages from the `messages_high` *queue*.

If that still doesn't make complete sense... don't worry: let's see this in action. First, make sure that your worker is *not* running: our's is stopped. Now let's go over and delete three photos - one, two, three - and upload four photos.

Cool! Let's see what happened in RabbitMQ! Inside the manager, click "Exchanges". Nice! We *do* have a new `messages_high_priority` exchange! The original `messages` exchange *still* sends all of its messages to a `messages` queue... but the new exchange sends all of *its* messages to a queue called `messages_high`. That's thanks to our `queues` config.

And... what's inside each queue? Go check it out! It's *exactly* what we want: the 3 deleted messages are waiting in the `messages` queue and the 4 newly-uploaded photos are in `messages_high`. Each transport is *successfully* getting their messages into a separate queue! And *that* means that we can consume them independently.

At the command line, we would normally tell Messenger to consume from `async_priority_high` and then `async` to get our prioritized delivery. But to *clearly* show what's happening, let's consume them independently for now. Start by consuming messages from the `async` transport:

```
php bin/console messenger:consume -vv async
```

It starts processing the `ImagePostDeletedEvent` objects... and stops after those three. It's done! That queue is empty. The command did *not* read the messages from `messages_high`. To do that, consume the `async_priority_high` transport:

```
php bin/console messenger:consume -vv async_priority_high
```

There we go! The *simplest*... but not fanciest... way to have prioritized transports with AMQP is to send each transport to a different exchange and configure it to route to a different queue. Later... we'll see the fancier way.

Before we get there, remember when I had you comment-out the `DelayStamp` before we started using RabbitMQ? Next, I'll show you why: we'll re-add that `DelayStamp` and see the *crazy* way that messages are "delayed" in RabbitMQ.

# Chapter 38: Delaying in AMQP: Dead Letter Exchange

When we started working with AMQP, I told you to go into `ImagePostController` and remove the `DelayStamp`. This stamp is a way to tell the transport system to *wait* at least 500 milliseconds before allowing a worker to receive the message. Let's change this to 10 seconds - so `10000` milliseconds.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             new DelayStamp(10000)
66         ]);
↕ // ... lines 67 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Now, move over to your terminal and make sure that your worker is *not* running.

Ok, let's see what happens! Right now *both* queues are empty. I'll upload 3 photos... then... quick, quick, quick! Go look at the queues. Suddenly, poof! A new queue appeared... with a strange name: `delay_messages_high_priority__10000`. And it has - dun, dun, dun! - *three* messages in it.

Let's look inside. Interesting, the messages were delivered *here*, instead of the normal queue. But then... they disappeared? The graph shows how the messages sitting in this queue went from 3 to 0. But... how? Our worker isn't even running!

Woh! This page just 404'ed! The queue is gone! Something is attacking our queues!

Head back to the queue list. Yea, that weird "delay" queue *is* gone... oh, but now the three messages are somehow in `messages_high`. What the heck just happened?

Well first, to prove that the whole system *still* works... regardless of what craziness just occurred... let's run our worker and consume from both the `async_priority_high` and `async` transports:

```
php bin/console messenger:consume -vv async_priority_high async
```

It consumes them and... when we move over, go to the homepage and refresh, yep! Ponka *was* added to those images.

## The Delay Exchange

Ok, let's figure out how this worked. I mean, on the one hand, it's not important: if we had been running our worker the entire time, you would have seen that those messages *were* in fact delayed by 10 seconds. *How* you delay messages in RabbitMQ is kinda crazy... but if you don't care about the details, Messenger just takes care of it for you.

But I *do* want to see how this works... in part because it'll be a *great* chance to see how some of the more advanced features of AMQP work.

Click on "Exchanges". Surprise! There's a *new* exchange called `delays`. And instead of being a `fanout` type like our other two exchanges, this is a `direct` exchange. We'll talk about what that that means soon.

But the *first* thing to know is that when Messenger sees that a message should be delayed, it sends it to *this* exchange *instead* of sending it to the normal, "correct" exchange. At this moment, the `delays` exchange has *no* bindings... but that will change when we send a delayed message.

To be able to *really* see what's happening, let's increase the delay to 60 seconds.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             new DelayStamp(60000)
66         ]);
↕ // ... lines 67 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

Ok, upload 3 more photos: we *now* know that these were just *sent* to the `delays` exchange. And... if you refresh that exchange... it has a new binding! This says:

> *"If a message sent here has a "routing key" set to `delay_messages_high_priority__60000`, then I will send that message to a queue called delay_messages_high_priority__60000"*

A "routing key" is an extra property that you can set on a message that's sent to AMQP. Normally Messenger doesn't set *any* routing key, but when a message has a *delay*, it *does*. And thanks to this binding - those three messages are sent to the `delay_messages_high_priority__60000` queue. This is how a `direct` exchange works: instead of sending each message to *all* queues bound to it, it uses the "binding key" rules to figure out which queue - or *queues* - a message should go to.

## Delay Queues: x-message-ttl and x-deal-letter-exchange

Click into the queue because it's *super* interesting. It has a few important properties. The first is an `x-message-ttl` set to 60 seconds. What does that means? When you set this on a queue, it means that, after a message has been sitting in this queue for 60 seconds, RabbitMQ should remove it... which seems crazy, right? Why would we want messages to only live for 60 seconds... and then be deleted? Well... it's by design... and works together with this second important property: `x-dead-letter-exchange`.

If a queue has this property, it tells Rabbit that when a message hits its 60 second TTL and needs to be removed, it should *not* be deleted. Instead, it should be *sent* to the `messages_high_priority` exchange.

So, Messenger delivers messages to the `delays` exchange with a routing key that makes it get sent here. Then, after sitting around for 60 seconds, the message is removed from this queue and sent to the `messages_high_priority` exchange. Yep, it's delivered to the correct place after 60 seconds!

And then... 404! Even the queue itself is marked as "temporary": once it doesn't have any messages left, it deletes itself.

When you click back to see the Queues, the messages *were* delivered to the `messages_high` queue... but that's already empty because our worker consumed them.

So... yea... wow! Whenever we publish a message with a delay, Messenger sets *all* of this up: it creates the temporary delay queue with the TTL and dead letter exchange settings, adds a binding to the `delays` exchange to route to this queue, and adds the correct routing key to the message to make sure it ends up in that queue.

You can *really* start to see how *rich* the features are in AMQP... even if you won't need them. The most important feature we just saw was the `direct` exchange type: an exchange that relies on routing keys to figure out where each message should go.

Next, could we use direct exchanges for our *non-delayed* messages? Instead of two exchanges that each "fan out" to a separate queue, could we create just *one* exchange that, by using routing keys, delivers the correct messages to the correct queues? Totally.

# Chapter 39: Exchange Routing and Binding Keys

Let's change this delay back to one second... so we're not waiting all day for our photos to be processed.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 23
24 class ImagePostController extends AbstractController
25 {
↕ // ... lines 26 - 40
41     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, MessageBusInterface $messageBus)
42     {
↕ // ... lines 43 - 63
64         $envelope = new Envelope($message, [
65             new DelayStamp(1000)
66         ]);
↕ // ... lines 67 - 69
70     }
↕ // ... lines 71 - 98
99 }
```

## Simple Setup: 1 Fanout Exchange per Queue

In `messenger.yaml`, the messages sent to each transport - `async` and `async_priority_high` - need to ultimately be delivered into two different queues so that we can consume them independently. And... we've accomplished that!

But there are two different ways that we could have done this. First, remember that in AMQP, messages are sent to an *exchange*, not a queue. Right now, when a message is routed to the `async` transport, Messenger sends that to an exchange called `messages`. You don't see that config here only because `messages` is the default exchange name in Messenger.

When a message is routed to the `async_priority_high` transport, Messenger sends that to an exchange called `messages_high_priority`. Each transport always sends to exactly *one* exchange.

Then, each exchange routes *every* message to a single queue, like the `messages` exchange sends to a `messages` queue... and `messages_high_priority` sends to a `messages_high` queue. There is *not* a routing key on the binding: Messenger binds each exchange to *one* queue... but with *no* routing key. That's how a "fanout" exchange works: it doesn't care about routing keys... it just sends each message to *every* queue bound to it.

## 1 Direct Exchange to 2 Queues

So that's *one* way to to solve this problem. The *other* way involves having only a *single* exchange... but making it smart enough to send some messages to the `messages` queue and other messages to `messages_high`. We do that with smarter binding and routing keys... which we already saw with the `delays` exchange.

## Configuring a Direct Exchange

Let's refactor our transports to use this "smarter" system. Under the `async` transport, add `options`, then `exchange`, and set `name` to `messages`. If we stopped here, this would change *nothing*: this is the default exchange name in Messenger.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... line 21
            async:
// ... lines 23 - 25
                options:
// ... line 27
                    exchange:
                        name: messages
// ... lines 30 - 53
```

But now, add a `type` key set to `direct`. This *does* change things: the default value is `fanout`. Add one more key below this: `default_publish_routing_key` set to `normal`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... line 21
            async:
// ... lines 23 - 25
                options:
// ... line 27
                    exchange:
                        name: messages
                        type: direct
                        default_publish_routing_key: normal
// ... lines 32 - 53
```

I'll talk about that in a second. Next, add a `queues` section. Let's "bind" this exchange to a queue called `messages_normal`. But we won't stop there! Under this, add `binding_keys` set to `[normal]`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... line 21
            async:
// ... lines 23 - 25
                options:
// ... line 27
                    exchange:
                        name: messages
                        type: direct
                        default_publish_routing_key: normal
// ... lines 32 - 33
                    queues:
                        messages_normal:
                            binding_keys: [normal]
// ... lines 37 - 53
```

That word `normal` could be *any* string. But it's no accident that this matches what we set for `default_publish_routing_key`.

## Deleting all the Exchanges and Queues

Instead of talking a ton about what this will do... let's... see it in action! Click to delete a photo: that should send a message to the `async` transport. Oh, but the AJAX call explodes! Open up the profiler to see the error. Ah:

> *"Server channel error: 406, message: PRECONDITION_FAILED - inequivalent arg 'type' for exchange 'messages': received 'direct' but current is 'fanout'"*

The problem is that we already have an exchange called `messages`, which is a `fanout` type... but now we're trying to use it as a `direct` exchange. AMQP is warning us that we're trying to do something crazy!

So let's start over. Now that we're doing things a *new* way, let's hit the reset button and allow Messenger to create everything new.

Find your terminal - I'll log out of MySQL - and stop your worker... otherwise it will *keep* trying to create your exchanges and queues with the old config.

Then move back to the RabbitMQ admin, delete the `messages` exchange... then the `messages_high_priority` exchange. And even though the queues won't look different, to be extra safe, let's delete both of them too.

So we're back to no queues and only the original exchanges that AMQP created - which we're not using anyways - and the `delays` exchange. We're starting from scratch!

Go back to our site, delete the second image and... it looks like it worked! Cool! Let's see what happened inside RabbitMQ! Yea! We have a new exchange called `messages` and it's a *direct* type. Inside, it has a *single* binding that says:

> *"When a message is sent to this exchange with a routing key called `normal`, it will be delivered to the queue called `messages_normal`."*

This was *all* set up thanks to the `queues` and `binding_keys` config. This tells Messenger:

> *"I want you to create a queue called `messages_normal`. Also, make sure that there is a binding on the exchange that will route any messages with a routing key set to `normal` to this queue."*

But... did Messenger *send* the message with that routing key? Until now, other than the delay stuff, Messenger has been delivering our messages to AMQP with *no* routing key. The

`default_publish_routing_key` config changes that. It says:

> *“Hey! Whenever a message is routed to the `async` transport, I want you to send it to the `messages` exchange with a routing key set to `normal`.”*

This *all* means that if we look at the queues... yep! We have a `message_normal` queue with *one* message waiting inside! We did it!

Next, let's repeat this for the *other* transport. Then, we'll learn how this gives us the flexibility to dynamically control where a message will be delivered at the moment we dispatch it.

# Chapter 40: Dynamic AMQP Routing Key (AmqpStamp)

Let's repeat the new exchange setup for the `async_priority_high` transport: we want this to deliver to the *same* direct exchange, but then use a different routing key to route messages to a different queue.

Change the exchange to `messages`, set the type to `direct`, then use `default_publish_routing_key` to automatically attach a routing key called `high` to each message.

Below, for the `messages_high` queue, this tells Messenger that we want this queue to be created and bound to the exchange. That's cool, but we *now* need that binding to have a routing key. Set `binding_keys` to `[high]`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... lines 21 - 37
            async_priority_high:
// ... line 39
                options:
                    exchange:
                        name: messages
                        type: direct
                        default_publish_routing_key: high
                    queues:
                        messages_high:
                            binding_keys: [high]
// ... lines 48 - 56
```

How can we trigger Messenger to create that new queue and add the new binding? Just perform *any* operation that uses this transport... like uploading a photo! Ok, go check out the RabbitMQ manager - start with Exchanges.

Yep, we still have just one `messages` exchange... but now it has two bindings! If you send a message to this exchange with a `high` routing key, it will be sent to `message_high`.

Click "Queues" to see... nice - a new `messages_high` queue with one message waiting inside.

And... we're done! This new setup has the same end-result: each transport ultimately delivers messages to a different queue. Let's go consume the waiting messages: consume `async_priority_high` then `async`.

```
php bin/console messenger:consume -vv async_priority_high async
```

And it consumes them in the correct order: handling `AddPonkaToImage` first because that's in the high priority queue and *then* moving to messages in the other queue.

By the way, when we consume from the `async` transport, for example, behind-the-scenes, it means that Messenger is *reading* messages from any queue that's configured for that transport. In our app, each transport has config for only one queue, but you *could* configure *multiple* queues under a transport and even set different binding keys for each one. But when you *consume* that transport, you'll be consuming messages from *every* queue you've configured.

## Dynamic Routing Keys

So, let's back up and look at the whole flow. When we dispatch an `AddPonkaToImage` object, our Messenger routing config *always* routes this to the `async_priority_high` transport. This causes the message to be sent to the `messages` exchange with a routing key set to `high`... and the binding logic means that it will ultimately be delivered to the `messages_high` queue.

Due to the way that Messenger's routing works - the fact that you route a *class* to a transport - every message *class* will *always* be delivered to the same queue. But what if you *did* want to control this dynamically? What if, at the moment you *dispatch* a message, you needed to send that message to a *different* transport than normal? Maybe you decide that *this* particular `AddPonkaToImage` message is *not* important and should be routed to `async`.

Well... that's just *not* possible with Messenger: each class is *always* routed to a specific transport. But this end-result *is* possible... if you know how to leverage routing keys.

Here's the trick: what if we could publish an `AddPonkaToImage` object... but tell Messenger that when it sends it to the exchange, it should use the `normal` routing key instead of `high`? Yea, the message would *technically* still be routed to the `async_priority_high` transport... but it would ultimately end up in the `messages_normal` queue. That would do it!

Is that possible? Totally! Open up `ImagePostController` and find where we dispatch the message. After the `DelayStamp`, add a new `AmqpStamp` - but be careful not to choose `AmqpReceivedStamp` - that's something different... and isn't useful for us. This stamp accepts a few arguments and the first one - gasp! - is the routing key to use! Pass this `normal`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 18
19 use Symfony\Component\Messenger\Transport\AmqpExt\AmqpStamp;
↕ // ... lines 20 - 24
25 class ImagePostController extends AbstractController
26 {
↕ // ... lines 27 - 41
42     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
43     {
↕ // ... lines 44 - 64
65         $envelope = new Envelope($message, [
66             new DelayStamp(1000),
↕ // ... lines 67 - 68
69             new AmqpStamp('normal')
70         ]);
↕ // ... lines 71 - 73
74     }
↕ // ... lines 75 - 102
103 }
```

Let's try it! Stop the worker so we can see what happens internally. Then, upload a photo, go to the RabbitMQ manager, click on queues... refresh until you see the message in the right queue... we have to wait for the delay... and there it is! It ended up in `messages_normal`.

## What else can you Customize on an Amqp Message?

By the way, if you look inside this `AmqpStamp` class, the second and third arguments are for something called `$flags` and `$attributes`. These are a bit more advanced, but might just

come in handy. I'll hit Shift+Shift to open a file called `Connection.php` - make sure to open the one in the `AmqpExt` directory. Now search for a method called `publishOnExchange()`.

When a message is sent to RabbitMQ, *this* is the low-level method that actually *does* that sending. Those `$flags` and `$attributes` from the stamp are used here! Passed as the third and fourth arguments to some `$exchange->publish()` method. Hold Cmd or Ctrl and click to jump into that method.

Oh! This jumps us to a "stub" - a "fake" method & declaration... because this class - called `AMQPExchange` is *not* something you'll find in your `vendor/` directory. Nope, this class comes from the AMQP PHP extension that we installed earlier.

So, if you find that you *really* need to control something about *how* a message is published through this extension, you can do that with the `$flags` and `$attributes`. The docs above this do a nice job of showing you the options.

And... that's it for AMQP and RabbitMQ! Sure, there's more to learn about RabbitMQ - it's a huge topic on its own - but you now have a firm grasp of its most important concepts and how they work. And unless you need to do some *pretty* advanced stuff, you understand *plenty* to work with Messenger.

Next, up until now we've been sending messages from our Symfony app *and* consuming them from that same app. But, that's not always the case. One of the powers of a "message broker" like RabbitMQ is the ability to send messages from one system and *handle* them in a totally different system... maybe on a totally different server or written in a totally different language. Craziness!

But if we're going to use Messenger to *send* messages to a queue that will then be handled by a totally different app... we probably need to encode those messages as JSON... instead of the PHP serialized format we're using now.

# Chapter 41: Serializing Messages as JSON

Once you start using RabbitMQ, a totally different workflow becomes possible... a workflow that's *especially* common with bigger systems. The idea is that the code that *sends* a message might *not* be the same code that consumes and *handles* that message. Our app is responsible for both sending the messages to RabbitMQ *and*, over here in the terminal, for *consuming* messages from the queue and handling them.

But what if we wanted to send one or more messages to RabbitMQ with the expectation that some *other* system - maybe some code written in a different language and deployed to a different server - will consume and handle it? How can we do that?

Well... on a high level... it's easy! If we wanted to send things to this `async` transport... but didn't plan to *consume* those messages, we wouldn't need to change anything in our code! Nope, we just... wouldn't consume messages from that transport when using the `messenger:consume` command. We could still consume messages from *other* transports - we just wouldn't read these ones... because we know someone else will. Done! Victory! Coffee!

## How are our Messages Formatted?

But... if you *were* going to send data to another system, how would you normally format that data? Well, to use a more familiar example, when you send data to an API endpoint, you typically format that data as JSON... or maybe XML. The same is true in the queueing world. You can send a message to RabbitMQ in *any* format... as long as whoever is consuming that message *understands* the format. So... what format are we using now? Let's find out!

I'll go into the `messages_normal` queue... and just to be safe, let's empty this. Messages sent to the `async` transport will eventually end up in this queue... and the `ImagePostDeleteEvent` classes route there. Ok, back on our app, delete a photo then, looking at our queue, in a moment... there it is! Our queue contains the one new message.

Let's see *exactly* what this message looks like. Down below, there's a spot to fetch a message out. But... for some reason... this hasn't been working for me. To hack around this, I'll bring up

my network tools, click "Get Message(s)" again... and look at the AJAX request this just made. Open up the returned data and hover over that `payload` property.

Yep, *this* is what our message looks like in the queue - this is the *body* of the message. What *is* that ugly format? It's a serialized PHP object! When Messenger consume this, it knows to use the `unserialize` function to get it back into an object... and so, this format works awesome!

But if we expect a *different* PHP application to consume this... unserializing it won't work because these classes probably won't exist. And if the code that will handle this is written in a different language, pfff, they won't even have a *chance* at reading and understanding this PHP-specific format.

The point is: using PHP serialization works *great* when the app that sends the message also handles it. But it works *horribly* when that's not the case. Instead, you'll probably want to use JSON or XML.

## Using the Symfony Serializer

Fortunately, using a different format is easy. I'll purge that message out of the queue one more time. Move over and open `config/packages/messenger.yaml`. One of the keys that you're allowed to have below each transport is called `serializer`. Set this to a special string: `messenger.transport.symfony_serializer`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... line 21
            async:
// ... line 23
                serializer: messenger.transport.symfony_serializer
// ... lines 25 - 57
```

When a message is sent to a transport - whether that's Doctrine, AMQP or something else - it uses a "serializer" to *encode* that message into a string format that can be sent. Later, when it *reads* a message from a transport, it uses that same serializer to *decode* the data back into the message object.

Messenger comes with two "serializers" out-of-the-box. The first one is the PHP serializer... which is the default. The second is the "Symfony Serializer", which uses Symfony's Serializer component. *That* is the serializer service that we just switched to. If you don't already have the serializer component installed, make sure you install it with:

```
composer require "serializer:^1.0"
```

The Symfony serializer is great because it's *really* good at turning objects into JSON or XML, and it uses JSON by default. So... let's see what happens! Move back and delete another photo. Back in the Rabbit manager, I'll use the same trick as before to see what that message looks like.

Woh. This is *fascinating*! The `payload` is now... *super* simple: just a `filename` key set to the filename. This is the JSON representation of the message class, which is `ImagePostDeletedEvent`. Open that up: `src/Message/Event/ImagePostDeletedEvent.php`. Yep! The Symfony serializer turned this object's *one* property into JSON.

We're not going to go *too* deep into Symfony's serializer component, but if you want to know more, we go *much* deeper in our API Platform Tutorial.

Anyways, this simple JSON structure *is* something *any* other system could understand. So... we rock!

But... just as a challenge... if we *did* try to *consume* this message from our Symfony app... would it work? I'm not sure. If this message is consumed, how would the serializer know that this simple JSON string needs to decoded into an `ImagePostDeletedEvent` object? The answer... lies somewhere else in the message: the headers. That's next.

# Chapter 42: JSON, Message Headers & Serializer Options

In addition to the payload, a message in RabbitMQ can also have "headers". Check that key out on our message. Woh! This contains a big JSON structure of the original class name *and* the data and class names of the *stamps* attached to the message!

Why did Messenger do this? Well, find your terminal and consume the `async` transport:

```
php bin/console messenger:consume -vv async
```

This *still* works. Internally, the Symfony serializer uses the info on the `headers` to figure out how to take this simple JSON string and turn it into the correct object. It used the `type` header to know that the JSON should become an `ImagePostDeletedEvent` object and then looped over the stamps and turned each of *those* back into a stamp object for the envelope.

The *really* nice thing about using the Symfony serializer in Messenger is that the `payload` is this simple, pure JSON structure that can be consumed by any application in any language. It *does* contain some PHP class info on the *headers*, but another app can just ignore that. But *thanks* to those headers, if the same app *does* both send and consume a message, the Symfony serializer can still be used.

## Shouldn't we Always use the Symfony Serializer?

But wait... if that's true - if the Symfony serializer creates messages that can be consumed by external systems *or* by our same app - then why isn't it the default serializer in Messenger? An *excellent* question! The reason is that the Symfony serializer requires your classes to follow a few *rules* in order to be serialized and unserialized correctly - like each property needs a setter method or a constructor argument where the name matches the property name. If your class doesn't follow those rules, you can end up with a property that is set on the original object, but suddenly becomes null when it's read from the transport. No fun.

In other words, the PHP serializer is easier and more dependable when everything is done by the same app.

## Configuring the Symfony Serializer

Anyways, if you *are* using the Symfony serializer, there are also a few things that can be configured. Find your terminal and run:

```
php bin/console config:dump framework messenger
```

Check out that `symfony_serializer` key. *This* is where you configure the behavior of the serializer: the format - `json`, `xml` or something else, and the `context`, which is an array of options for the serializer.

Of course, you can *also* create a totally *custom* serializer service. And if you have the *opposite* workflow to what we just described - one where your app *consumes* messages that were sent to Rabbit from some *other* system - a custom serializer is *exactly* what you need. Let's talk about that next.

# Chapter 43: Setup for Messages from an Outside System

What if a queue on RabbitMQ was filled with messages that originated from an *external* system... but we wanted to consume and *handle* those from our Symfony App? For example, maybe a user can request that a photo be deleted from some totally different system... and that system needs to communicate *back* to our app so that it can actually *do* the deleting? How would that work?

Each transport in Messenger really has *two* jobs: one, to send messages to a message broker or queue system and two, to *receive* messages from that same system and handle them.

And, like we talked about in the last video, you don't need to use *both* features of a transport: you could choose to *send* to a transport, but never read and *consume* those messages... because some other system will. Or, you can do the opposite: create a transport that you will never *send* to, but that you *will* use to *consume* messages... that were probably put there by some outside system. The *trick* to doing this is creating a serializer that can understand the *format* of those outside messages.

## Creating a new Message & Handler

Instead of over-explaining this, let's see it in action. First, pretend that this imaginary external system needs to be able to tell our app to do something... very... important: to log an Emoji. Ok, this may not be the *most* impressive type of a message... but the details of *what* this outside message is telling our app to do aren't important: it could be telling us to upload an image with details about where the file is located, delete an image, send an email to a registered user or, log an emoji!

Let's get this set up. Normally, if *we* wanted to dispatch a command to log an Emoji, we would start by creating a message class and message handler. In this case... we'll do the *exact* same thing. In the `Command/` directory, create a new PHP class called `LogEmoji`.

src/Message/Command/LogEmoji.php

```
↕ // ... lines 1 - 2
3 namespace App\Message\Command;
4
5 class LogEmoji
6 {
↕ // ... lines 7 - 17
18 }
```

Add a `public function __construct()`. In order to tell us *which* emoji to log, the outside system will send us an integer *index* of the emoji they want - our app will have a list of emojis. So, add an `$emojiIndex` argument and then press Alt+Enter and select "Initialize Fields" to create that property and set it.

src/Message/Command/LogEmoji.php

```
↕ // ... lines 1 - 2
3 namespace App\Message\Command;
4
5 class LogEmoji
6 {
7     private $emojiIndex;
8
9     public function __construct(int $emojiIndex)
10    {
11        $this->emojiIndex = $emojiIndex;
12    }
↕ // ... lines 13 - 17
18 }
```

To make this property *readable* by the handler, go to the Code -> Generate menu - or Command + N on a Mac - select getters and generate `getEmojiIndex()`.

src/Message/Command/LogEmoji.php

```
// ... lines 1 - 2
namespace App\Message\Command;

class LogEmoji
{
    private $emojiIndex;

    public function __construct(int $emojiIndex)
    {
        $this->emojiIndex = $emojiIndex;
    }

    public function getEmojiIndex(): int
    {
        return $this->emojiIndex;
    }
}
```

Brilliant! A *perfectly* boring, um, normal, message class. Step two: in the `MessageHandler/Command/` directory, create a new `LogEmojiHandler` class. Make this implement our normal `MessageHandlerInterface` and add `public function __invoke()` with the type-hint for the message: `LogEmoji $logEmoji`.

src/MessageHandler/Command/LogEmojiHandler.php

```
// ... lines 1 - 2
namespace App\MessageHandler\Command;

use App\Message\Command\LogEmoji;
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;

class LogEmojiHandler implements MessageHandlerInterface
{
    public function __invoke(LogEmoji $logEmoji)
    {
    }
}
```

Now... we get to work! I'll paste an emoji list on top: here are the five that the outside system can choose from: cookie, dinosaur, cheese, robot, and of course, poop.

src/MessageHandler/Command/LogEmojiHandler.php

```
↕ // ... lines 1 - 8
9  class LogEmojiHandler implements MessageHandlerInterface
10 {
11     private static $emojis = [
12         '?',
13         '?',
14         '?',
15         '?',
16         '?'
17     ];
↕ // ... lines 18 - 33
34 }
```

And then, because we're going to be logging something, add an `__construct()` method with the `LoggerInterface` type hint. Hit Alt + Enter and select "Initialize Fields" one more time to create *that* property and set it.

src/MessageHandler/Command/LogEmojiHandler.php

```
↕ // ... lines 1 - 5
6  use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 8
9  class LogEmojiHandler implements MessageHandlerInterface
10 {
11     private static $emojis = [
12         '?',
13         '?',
14         '?',
15         '?',
16         '?'
17     ];
18
19     private $logger;
20
21     public function __construct(LoggerInterface $logger)
22     {
23         $this->logger = $logger;
24     }
↕ // ... lines 25 - 33
34 }
```

Inside `__invoke()`, our job is pretty simple. To get the emoji, set an `$index` variable to `$logEmoji->getEmojiIndex()`.

src/MessageHandler/Command/LogEmojiHandler.php

```
↕ // ... lines 1 - 5
6 use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 8
9 class LogEmojiHandler implements MessageHandlerInterface
10 {
11     private static $emojis = [
12         '?',
13         '?',
14         '?',
15         '?',
16         '?'
17     ];
18
19     private $logger;
20
21     public function __construct(LoggerInterface $logger)
22     {
23         $this->logger = $logger;
24     }
25
26     public function __invoke(LogEmoji $logEmoji)
27     {
28         $index = $logEmoji->getEmojiIndex();
29
↕ // ... lines 30 - 32
33     }
34 }
```

Then `$emoji = self::$emojis` - to reference that static property - `self::$emojis[$index] ?? self::emojis[0]`.

src/MessageHandler/Command/LogEmojiHandler.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 8
class LogEmojiHandler implements MessageHandlerInterface
{
    private static $emojis = [
        '?',
        '?',
        '?',
        '?',
        '?'
    ];

    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function __invoke(LogEmoji $logEmoji)
    {
        $index = $logEmoji->getEmojiIndex();

        $emoji = self::$emojis[$index] ?? self::$emojis[0];
// ... lines 31 - 32
    }
}
```

In other words, *if* the index exists, use it. Otherwise, fallback to logging a cookie... cause... everyone loves cookies. Log with `$this->logger->info('Important message! ')` and then `$emoji`.

src/MessageHandler/Command/LogEmojiHandler.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 8
class LogEmojiHandler implements MessageHandlerInterface
{
    private static $emojis = [
        '?',
        '?',
        '?',
        '?',
        '?'
    ];

    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function __invoke(LogEmoji $logEmoji)
    {
        $index = $logEmoji->getEmojiIndex();

        $emoji = self::$emojis[$index] ?? self::$emojis[0];

        $this->logger->info('Important message! '.$emoji);
    }
}
```

The *big* takeaway from this new message and message handler is that it is, well, absolutely *no* different from *any* other message and message handler! Messenger does *not* care whether the `LogEmoji` object will be dispatched manually from our own app or if a worker will receive a message from an outside system that will get mapped to this class.

To prove it, go up to `ImagePostController`, find the `create()` method and, *just* to see make sure this is working, add: `$messageBus->dispatch(new LogEmoji(2))`.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 7
8 use App\Message\Command\LogEmoji;
↕ // ... lines 9 - 25
26 class ImagePostController extends AbstractController
27 {
↕ // ... lines 28 - 42
43     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, MessageBusInterface $messageBus)
44     {
↕ // ... lines 45 - 73
74         $messageBus->dispatch(new LogEmoji(2));
↕ // ... lines 75 - 76
77     }
↕ // ... lines 78 - 105
106 }
```

If this *is* working, we should see a message in our logs each time we upload a photo. Find your terminal: let's watch the logs with:

```
tail -f var/log/dev.log
```

That's the log file for the `dev` environment. I'll clear my screen, then move over, select a photo and... move back. There it is:

> *"Important message!* 🧀*"*

I agree! That *is* important! This is cool... but not what we really want. What we *really* want to do is use a worker to consume a message from a queue - probably a JSON message - and *transform* that intelligently into a `LogEmoji` object so Messenger can handle it. How do we do that? With a dedicated transport and a customer serializer. Let's do that next!

# Chapter 44: Transport for Consuming External Messages

We've just created a new message class & handler... then instantiated it and dispatched it directly into the message bus. Yep, we just did something totally... boring! But... it's actually pretty similar to our *real* goal! Our *real* goal is to pretend that an *outside* system is putting messages into a RabbitMQ queue... probably formatted as JSON... and *we* will read those messages, transform that JSON into a `LogEmoji` object and... basically dispatch that through the message bus. It's really the same basic flow: in both cases, we create a `LogEmoji` object and pass it to Messenger.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 25
26 class ImagePostController extends AbstractController
27 {
↕ // ... lines 28 - 42
43     public function create(Request $request, ValidatorInterface $validator, PhotoFileManager $photoManager, EntityManagerInterface $entityManager, MessageBusInterface $messageBus)
44     {
↕ // ... lines 45 - 73
74         //$messageBus->dispatch(new LogEmoji(2));
↕ // ... lines 75 - 76
77     }
↕ // ... lines 78 - 105
106 }
```

## Creating a Dedicated Transport

The first step is to create a transport that will read these messages from whatever queue the outside system is placing them into. We'll keep the `async` and `async_priority_high` transports: we'll continue to send and receive from those. But now create a new one called, how about: `external_messages`. I'll use the same DSN because we're *still* consuming things from RabbitMQ. But for the options, instead of consuming messages from `message_high` or `messages_normal`, we'll consume them from whatever queue that outside system is using - let's pretend it's called `messages_from_external`. Set that to just `~`.

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... lines 21 - 50
            external_messages:
                dsn: '%env(MESSENGER_TRANSPORT_DSN)%'
                options:
                    exchange:
                        name: messages
                        type: direct
                        default_publish_routing_key: from_external
                    queues:
                        messages_from_external:
                            binding_keys: [from_external]
// ... lines 61 - 69
```

By the way, it *is* important that we use a *different* transport that reads from a *different* queue for these external messages. Why? Because, as you'll see in a few minutes, these external messages will need special logic to decode them back into the correct object. We'll attach that special logic to the transport.

Anyways, above this add `auto_setup: false`.

**Tip**

To support retry, you *should* use `auto_setup` and configure a few more things. See the tip below for more details.

Ok, there are a few important things happening here. The first is that this queue config means that when we *consume* from the `external_messages` transport, Messenger will read messages from a queue called `messages_from_external`. The second important thing is `auto_setup: false`. This tells Messenger *not* to try to create this queue. Why? Well... I guess our app *could* create that queue... that would probably be fine... but since we're expecting an external system to send messages to this queue, I'm guessing that *that* system will want to be responsible for making sure it exists.

Oh, and you probably also noticed that I didn't add any `exchange` config. That was on purpose. An exchange is only used when *sending* a message. And because we're not planning

on *ever* sending a message through this transport, that part of the transport just won't ever be used.

> **Tip**
>
> Correction: if you're using AMQP and want "retries" to work, you *will* need to configure a routing & binding key so that if a message needs to be sent to this transport (for redelivery), Messenger can attach the correct binding key so that the message will end up in the `messages_from_external` queue. See the code block on this page for an updated example.

So with *just* this, we should be able to consume from the new transport. Spin over to your terminal and run:

```
php bin/console messenger:consume -vv external_messages
```

And... it explodes! This is awesome.

> *"Server channel error: 404, message: NOT_FOUND - no queue 'messages_from_external'"*

We're seeing our `auto_setup: false` in action! Instead of creating that queue when it didn't exist, it exploded. Love it!

## Creating the Queue By Hand

So now, let's pretend that *we* are that "external" system and *we* want to create that queue. Copy the queue name - `messages_from_external` - and, inside the Rabbit Manager, create a new queue with that name. Don't worry about the options - they won't matter for us.

And... hello queue! Let's go see if we can consume messages from it:

```
php bin/console messenger:consume -vv external_messages
```

It works! Well... there aren't any *messages* in the queue yet, but it's happily checking for them.

## Putting an "External" Message into the queue

*Now*, let's *continue* to pretend like *we* are the "external" system that will be sending messages to this queue. On the queue management screen, we can publish a message into the queue. Convenient!

So... what will these messages look like? Well... they can *look* like anything: JSON, XML, a binary image, ASCII art - whatever we want. We'll just need to make sure that our Symfony app can *understand* the message - that's something we'll work on in a few minutes.

Let's think: if an outside system wants to send our app a *command* to log an emoji... and it can choose *which* emoji via a number... then... maybe the message is JSON that looks like this? An `emoji` key set to 2:

```
{
  "emoji": 2
}
```

Publish! Ok, go check the worker! Woh... it exploded! Cool!

> *“Could not decode message using PHP serialization”*

And then it shows our JSON. Of course! If you're consuming a message that was placed in the queue by an external system... that message *probably* won't be in the PHP serialized format... and it really *shouldn't* be. Nope, the message will probably be JSON or XML. The problem is that our transport is trying to transform that JSON into an object by using the default PHP serializer. Literally, it's calling `unserialize()` on that JSON.

We need to be smarter: when a transport consumes messages from an external system, it needs to have a *custom* serializer so we can take control. Let's do that next.

# Chapter 45: Custom Transport Serializer

If an external system sends messages to a queue that we're going to read, those messages will probably be sent as JSON or XML. We added a message formatted as JSON. To read those, we set up a transport called `external_messages`. But when we consumed that JSON message... it exploded! Why? Because the *default* serializer for every transport is the `PhpSerializer`. Basically, it's trying to call `unserialize()` on our JSON. That's...uh... not gonna work.

Nope, if you're consuming messages that came from an external system, you're going to need a custom serializer for your transport. Creating a custom serializer is... actually a *very* pleasant experience.

## Creating the Custom Serializer Class

Inside of our `src/Messenger/` directory... though this class could live anywhere.. let's create a new PHP class called `ExternalJsonMessengerSerializer`. The only rule is that this needs to implement `SerializerInterface`. But, careful! There are *two* `SerializerInterface`: one is from the Serializer component. We want the *other* one: the one from the Messenger component. I'll go to the "Code Generate" menu - or Command + N on a Mac - and select "Implement Methods" to add the two that this interface requires: `decode()` and `encode()`.

src/Messenger/ExternalJsonMessageSerializer.php

```php
// ... lines 1 - 2
namespace App\Messenger;

use Symfony\Component\Messenger\Envelope;
use Symfony\Component\Messenger\Exception\MessageDecodingFailedException;
use
Symfony\Component\Messenger\Transport\Serialization\SerializerInterface;

class ExternalJsonMessageSerializer implements SerializerInterface
{
    public function decode(array $encodedEnvelope): Envelope
    {
        // TODO: Implement decode() method.
    }

    public function encode(Envelope $envelope): array
    {
        // TODO: Implement encode() method.
    }
}
```

## The encode() Method

The idea is beautifully simple: when we *send* a message through a transport that uses this serializer, the transport will call the `encode()` method and pass us the `Envelope` object that contains the message. Our job is to turn that into a string format that can be sent to the transport. Oh, well, notice that this returns an *array*. But if you look at the `SerializerInterface`, this method should return an array with two keys: `body` - the body of the message - and `headers` - any headers that should be sent.

Nice, right? But... we're actually *never* going to *send* any messages through our external transport... so we don't need this method. To prove that it will never be called, throw a new `Exception` with:

> *"Transport & serializer not meant for sending messages"*

src/Messenger/ExternalJsonMessageSerializer.php

```
// ... lines 1 - 9
class ExternalJsonMessageSerializer implements SerializerInterface
{
// ... lines 12 - 22
    public function encode(Envelope $envelope): array
    {
        throw new \Exception('Transport & serializer not meant for sending messages');
    }
}
```

That'll give me a gentle reminder in case I do something silly and route a message to a transport that uses this serializer by accident.

**Tip**

Actually, if you want your messages to be redelivered, you *do* need to implement the `encode()` method. See the code-block on this page for an example, which includes a small update to `decode()`.

src/Messenger/ExternalJsonMessageSerializer.php

```php
// ... lines 1 - 9
class ExternalJsonMessageSerializer implements SerializerInterface
{
    public function decode(array $encodedEnvelope): Envelope
    {
// ... lines 14 - 19
        // in case of redelivery, unserialize any stamps
        $stamps = [];
        if (isset($headers['stamps'])) {
            $stamps = unserialize($headers['stamps']);
        }

        return new Envelope($message, $stamps);
    }

    public function encode(Envelope $envelope): array
    {
        // this is called if a message is redelivered for "retry"
        $message = $envelope->getMessage();

        // expand this logic later if you handle more than
        // just one message class
        if ($message instanceof LogEmoji) {
            // recreate what the data originally looked like
            $data = ['emoji' => $message->getEmojiIndex()];
        } else {
            throw new \Exception('Unsupported message class');
        }

        $allStamps = [];
        foreach ($envelope->all() as $stamps) {
            $allStamps = array_merge($allStamps, $stamps);
        }

        return [
            'body' => json_encode($data),
            'headers' => [
                // store stamps as a header - to be read in decode()
                'stamps' => serialize($allStamps)
            ],
        ];
    }
}
```

## The decode() Method

The method that *we* need to focus on is `decode()`. When a worker consumes a message from a transport, the transport calls `decode()` on its serializer. Our job is to read the message from the queue and turn that into an `Envelope` object with the *message* object inside. If you check out the `SerializerInterface` one more time, you'll see that the argument we're passed - `$encodedEnvelope` - is really just an array with the same two keys we saw a moment ago: `body` and `headers`.

Let's separate the pieces first: `$body = $encodedEnvelope['body']` and `$headers = $encodedEnvelope['headers']`. The `$body` will be the raw JSON in the message. We'll talk about the headers later: it's empty right now.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 9
10 class ExternalJsonMessageSerializer implements SerializerInterface
11 {
12     public function decode(array $encodedEnvelope): Envelope
13     {
14         $body = $encodedEnvelope['body'];
15         $headers = $encodedEnvelope['headers'];
↕ // ... lines 16 - 20
21     }
↕ // ... lines 22 - 26
27 }
```

## Turning JSON into the Envelope

Ok, remember our goal here: to turn this JSON into a `LogEmoji` object and then put that into an `Envelope` object. How? Let's keep it simple! Start with `$data = json_decode($body, true)` to turn the JSON into an associative array.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 9
10 class ExternalJsonMessageSerializer implements SerializerInterface
11 {
12     public function decode(array $encodedEnvelope): Envelope
13     {
14         $body = $encodedEnvelope['body'];
15         $headers = $encodedEnvelope['headers'];
16
17         $data = json_decode($body, true);
↕ // ... lines 18 - 20
21     }
↕ // ... lines 22 - 26
27 }
```

I'm not doing any error-checking yet... like to check that this is *valid* JSON - we'll do that a bit later. Now say: `$message = new LogEmoji($data['emoji'])` because `emoji` is the key in the JSON that we've decided will hold the `$emojiIndex`.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 4
5 use App\Message\Command\LogEmoji;
↕ // ... lines 6 - 9
10 class ExternalJsonMessageSerializer implements SerializerInterface
11 {
12     public function decode(array $encodedEnvelope): Envelope
13     {
14         $body = $encodedEnvelope['body'];
15         $headers = $encodedEnvelope['headers'];
16
17         $data = json_decode($body, true);
18         $message = new LogEmoji($data['emoji']);
↕ // ... lines 19 - 20
21     }
↕ // ... lines 22 - 26
27 }
```

Finally, we need to return an `Envelope` object. Remember: an `Envelope` is just a small wrapper *around* the message itself... and it might also hold some stamps. At the bottom, return `new Envelope()` and put `$message` inside.

src/Messenger/ExternalJsonMessageSerializer.php

```
// ... lines 1 - 4
use App\Message\Command\LogEmoji;
use Symfony\Component\Messenger\Envelope;
// ... lines 7 - 9
class ExternalJsonMessageSerializer implements SerializerInterface
{
    public function decode(array $encodedEnvelope): Envelope
    {
        $body = $encodedEnvelope['body'];
        $headers = $encodedEnvelope['headers'];

        $data = json_decode($body, true);
        $message = new LogEmoji($data['emoji']);

        // in case of redelivery, unserialize any stamps
        $stamps = [];
        if (isset($headers['stamps'])) {
            $stamps = unserialize($headers['stamps']);
        }

        return new Envelope($message, $stamps);
    }
// ... lines 28 - 55
}
```

## Configuring the Serializer on the Transport

Done! We rock! This is already a *fully* functional serializer that can *read* messages from a queue. But our transport won't just start "magically" using it: we need to configure that. And.. we already know how! We learned earlier that each transport can have a `serializer` option. Below the external transport, add `serializer` and set this to the *id* of our service, which is the same as the class name: `App\Messenger\`... and then I'll go copy the class name: `ExternalJsonMessengerSerializer`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 19
        transports:
// ... lines 21 - 50
            external_messages:
// ... line 52
                serializer: App\Messenger\ExternalJsonMessageSerializer
// ... lines 54 - 69
```

*This* is why we created a separate transport with a separate queue: we *only* want the *external* messages to use our `ExternalJsonMessengerSerializer`. The other two transports - `async` and `async_priority_high` - will still use the simpler PhpSerializer... which is *perfect*.

Ok, let's try this! First, find an open terminal and tail the logs:

```
tail -f var/log/dev.log
```

And I'll clear the screen. Then, in my other terminal, I'll consume messages from the `external_messages` transport:

```
php bin/console messenger:consume -vv external_messages
```

Perfect! There are no messages yet... so it's just waiting. But we're *hoping* that when we publish this message to the queue, it will be consumed by our worker, decoded correctly, and that an emoji will be logged! Ah, ok - let's try it. Publish! Oh, then move back over to the terminal.... there it is! We got an important message: cheese: it received the message and handled it down here.

So... we did it! We *rock*!

But... when we created the `Envelope`, we didn't put any stamps into it. Should we have? Does a message that goes through the "normal" flow have some stamps on it that we should manually add here? Let's dive into the workflow of a message and its *stamps*, next.

# Chapter 46: The Lifecycle of a Message & its Stamps

Forget about asynchronous messages and external transports and all that stuff. Open up `ImagePostController`. As a reminder, when you dispatch a message, you *actually* dispatch an `Envelope` object, which is a simple "wrapper" that contains the message itself and *may* also contain some stamps... which add extra info.

If you dispatch the *message* object directly, the message bus creates an `Envelope` *for* you and puts your message inside. The point is, internally, Messenger is *always* working with an `Envelope`. And when you call `$messageBus->dispatch()`, it also *returns* an `Envelope`: the *final* `Envelope` after Messenger has done all its work.

Let's see what that looks like: `dump()` that whole `$messageBus->dispatch()` line. Now, move over and upload a photo. Once that's done, find that request on the web debug toolbar... and open the profiler.

src/Controller/ImagePostController.php

```
↕ // ... lines 1 - 25
26 class ImagePostController extends AbstractController
27 {
↕ // ... lines 28 - 42
43     public function create(Request $request, ValidatorInterface
   $validator, PhotoFileManager $photoManager, EntityManagerInterface
   $entityManager, MessageBusInterface $messageBus)
44     {
↕ // ... lines 45 - 71
72         dump($messageBus->dispatch($envelope));
↕ // ... lines 73 - 76
77     }
↕ // ... lines 78 - 105
106 }
```

## The Envelope & Stamps after Dispatching

Perfect! You can see that the *final* `Envelope` has the original message object inside: `AddPonkaToImage`. But this `Envelope` *now* has more *stamps* on it.

Quick review time! When we dispatch a message into the message bus, it goes through a collection of *middleware*... and each middleware can add extra *stamps* to the envelope. If you expand `stamps` in the dump, wow! There are now *5* stamps! The first two - `DelayStamp` and `AmqpStamp` - are no mystery. *We* added those manually when we originally dispatched the message. The *last* one - `SentStamp` - is a stamp that's added by the `SendMessageMiddleware`. Because we've configured this message to be routed to the `async_priority_high` transport, the `SendMessageMiddleware` *sends* the message to RabbitMQ and then adds this `SentStamp`. This is a *signal* - to anyone who cares - us, or other middleware - that this message *was* in fact "sent" to a transport. Actually, it's *thanks* to this stamp that the *next* middleware that executes - `HandleMessageMiddleware` - knows that it should *not* handle this message right now. It sees that `SentStamp`, realizes the message was sent to a transport and so, does nothing. It will be handled later.

## BusNameStamp: How the Worker Dispatches to the Correct Bus

But what about this `BusNameStamp`? Let's open up that class. Huh, `BusNameStamp` *literally* contains... the name of the bus that the message was dispatched into. If you look in `messenger.yaml`, at the top, we have *three* buses: `command.bus`, `event.bus` and `query.bus`. Ok, but what's the point of `BusNameStamp`? I mean, we *dispatched* the message through the command bus... so why is it important that the message has a stamp on it that says this?

The answer is all about what happens when a worker *consumes* this message. The process looks like this. First, the `messenger:consume` command - that's the "worker" - reads a message off of a queue. Second, that transport's serializer turns that into an `Envelope` object with a message object inside - like our `LogEmoji` object. Finally, the worker *dispatches* that Envelope back into the message bus! Yea, internally, something calls `$messageBus->dispatch($envelope)`!

Wait... but if we have *multiple* message buses... how does the worker know *which* message bus it should dispatch the Envelope into? Whelp! *That* is the *purpose* of this `BusNameStamp`.

Messenger adds this stamp so that when the worker *receives* this message, it can use the stamp to dispatch the message into the correct bus.

Right now, in our serializer, we're not adding *any* stamps to the `Envelope`. Because the stamp doesn't exist, the worker uses the `default_bus`, which is the `command.bus`. So, in this case... it guessed correctly! This message *is* a command.

## The UniqueIdStamp

The *last* stamp that was added was this `UniqueIdStamp`. This is something that *we* created... and it's added via a custom middleware: `AuditMiddleware`. Whenever a message is dispatched, this middleware makes sure that every `Envelope` has exactly *one* `UniqueIdStamp`. Then, anyone can use the unique id string on that stamp to track *this* exact message through the whole process.

Wait... so if this is *normally* added when we originally *dispatch* a message... should we manually add the stamp inside of our serializer so that the `Envelope` has one?

Look at it this way: a *normal* message that's *sent* from our app would *already* have this stamp by the time it's published to RabbitMQ. When a worker receives it, it'll be there.

But... in this case, as you can clearly see, after receiving the external message, we are *not* adding that stamp. So, is that something we should add here so this "acts" like other messages?

Great question! The answer is... no! Check out the log messages: you can already see some messages with this `5d7bc` string. *That* is the unique id. Our message *does* have a `UniqueIdStamp`!

How? Remember, after our serializer returns the `Envelope`, the worker dispatches it *back* through the bus. And so, our `AuditMiddleware` is called, it adds that stamp and then logs some messages about it.

## The Big Takeaways

To back up a bit, there are *two* big points I'm trying to make. First, when a message is read and handled via a worker, it is *dispatched* through the message bus and all the normal middleware

are executed. For a message that is both sent from our app *and* handled by our app, it will go through the middleware *two* times.

The second important point is that when you consume a message that was put there from an external system, that message *might* be missing some stamps that a normal message would have. And, for the most part, that's probably fine! The `DelayStamp` and `AmqpStamp` are irrelevant because those both tell the transport how to *send* the message.

## Adding the BusNameStamp

But... the `BusNameStamp` *is* one that you might want to add. Sure, Messenger used the correct bus in this case by accident, but we can be more explicit!

Head into `ExternalJsonMessengerSerializer`. Change this to `$envelope = new Envelope()` and, at the bottom, return `$envelope`. Add the stamp with `$envelope = $envelope->with()` - this is how you add a stamp - `new BusNameStamp()`.

Then... hmm... because our transport & serializer only handle this *one* message... and because this *one* message is a command, we'll want to put the command bus here. Copy the `command.bus` bus name and paste. I'll add a comment that says that this is *technically* only needed if you need the message to be sent through a non-default bus.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\Messenger\Stamp\BusNameStamp;
↕ // ... lines 9 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 26
27         $envelope = new Envelope($message, $stamps);
28
29         // needed only if you need this to be sent through the non-default
   bus
30         $envelope = $envelope->with(new BusNameStamp('command.bus'));
31
32         return $envelope;
↕ // ... lines 33 - 61
62 }
```

Next, our serializer is great, but we didn't code very defensively. What would happen if the message contained invalid JSON... or was missing the `emoji` field? Would our app fail gracefully... or explode?

# Chapter 47: Graceful Failure in the Transport Serializer

Our shiny new `external_messages` transport reads messages from this `messages_from_external` queue, which we're *pretending* is being populated by an external application. We're taking this JSON and, in `ExternalJsonMessengerSerializer`, decoding it, creating the `LogEmoji` object, putting it into an `Envelope`, even adding a *stamp* to it, and ultimately returning it, so that it can *then* be dispatched back through the message bus system.

## Failing on Invalid JSON

This is looking great! But there are two improvements I want to make. First, we haven't been coding very defensively. For example, what if, for some reason, the message contains invalid JSON? Let's check for that: if `null === $data`, then throw a `new MessageDecodingFailedException('Invalid JSON')`

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 6
7  use Symfony\Component\Messenger\Exception\MessageDecodingFailedException;
↕ // ... lines 8 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 19
20         if (null === $data) {
21             throw new MessageDecodingFailedException('Invalid JSON');
22         }
↕ // ... lines 23 - 34
35     }
↕ // ... lines 36 - 40
41 }
```

I'll show you why we're using this *exact* exception class in a minute. But let's try this with some invalid JSON and... see what happens. Go restart the worker so it sees our new code:

```
php bin/console messenger:consume -vv external_messages
```

Then, in the RabbitMQ manager, let's make a *very* annoying JSON mistake: add a comma after the last property. Publish that message! Ok, move over and... explosion!

> *"MessageDecodingFailedException: Invalid JSON"*

Oh, and interesting: this *killed* our worker process! Yep, if an error happens during the *decoding* process, the exception *does* kill your worker. That's not ideal... but in reality... it's not a problem. On production, you'll already be using something like supervisor that will *restart* the process when it dies.

## Failing on Missing JSON Field

Let's add code to check for a *different* possible problem: let's check to see if this `emoji` key is missing: if not `isset($data['emoji'])`, this time throw a *normal* exception: `throw new \Exception('Missing the emoji key!')`.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 23
24         if (!isset($data['emoji'])) {
25             throw new \Exception('Missing the emoji key!');
26         }
↕ // ... lines 27 - 34
35     }
↕ // ... lines 36 - 40
41 }
```

Ok, move over and restart the worker:

```
php bin/console messenger:consume -vv external_messages
```

Back in Rabbit, remove the extra comma and change `emoji` to `emojis`. Publish! Over in the terminal... great! It exploded! And other than the exception *class*... it looks *identical* to the failure we saw before:

> *"Exception: Missing the emoji key!"*

But... something different *did* just happen. Try running the worker again:

```
php bin/console messenger:consume -vv external_messages
```

Woh! It exploded! Missing the emoji key. Run it again:

```
php bin/console messenger:consume -vv external_messages
```

## The Magic of MessageDecodingFailedException

The same error! *This* is the difference between throwing a normal `Exception` in the serializer versus the special `MessageDecodingFailedException`. When you throw a `MessageDecodingFailedException`, your serializer is basically saying:

> *"Hey! Something went wrong... and I do want to throw an exception. But, I think we should discard this message from the queue: there is no point to trying it over and over again. kthxbai!"*

And that's *super* important. If we don't discard this message, each time our worker restarts, it will fail on that *same* message... over-and-over again... forever. Any *new* messages will start piling up *behind* it in the queue.

So let's change the `Exception` to `MessageDecodingFailedException`. Try it now:

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 23
24         if (!isset($data['emoji'])) {
25             throw new MessageDecodingFailedException('Missing the emoji
   key!');
26         }
↕ // ... lines 27 - 34
35     }
↕ // ... lines 36 - 40
41 }
```

```
php bin/console messenger:consume -vv external_messages
```

It will explode the first time... but the `MessageDecodingFailedException` *should* have removed it from the queue. When we run the worker now:

```
php bin/console messenger:consume -vv external_messages
```

Yep! The message is gone and the queue is empty.

Next, let's add *one* more superpower to this serializer. What if that outside system actually sends our app *many* different types of message - not only a message to log emojis, but maybe also messages to delete photos or cook some pizza! How can our serializer figure out which messages are which... and which message *object* to create?

# Chapter 48: Mapping Messages to Classes in a Transport Serializer

We've written our transport serializer to *always* expect only *one* type of message to be put into the queue: a message that tells our app to "log an emoji". Your app *might* be that simple, but it's more likely that this "external" system might send 5 or 10 different *types* of messages. In that case, our serializer needs to detect which *type* of message this is and then turn it into the correct message *object*.

How can we do that? How can we figure out which *one* type of message this is? Do we... just look at what fields the JSON has? We *could*... but we can also do something smarter.

## Refactoring to a switch

Let's start by reorganizing this class a bit. Select the code at the bottom of this method - the stuff related to the `LogEmoji` object - and then go to the Refactor -> "Refactor This" menu, which is Ctrl+T on a Mac. Refactor this code to a method called `createLogEmojiEnvelope`.

**Tip**

To make sure "retries" work correctly, some of the code in this section has been tweaked. See the code blocks on this page for the updated examples!

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 23
24         $envelope = $this->createLogEmojiEnvelope($data);
25
26         // in case of redelivery, unserialize any stamps
27         $stamps = [];
28         if (isset($headers['stamps'])) {
29             $stamps = unserialize($headers['stamps']);
30         }
31         $envelope = $envelope->with(... $stamps);
32
33         return $envelope;
34     }
↕ // ... lines 35 - 63
64     private function createLogEmojiEnvelope($data): Envelope
65     {
66         if (!isset($data['emoji'])) {
67             throw new MessageDecodingFailedException('Missing the emoji
   key!');
68         }
69         $message = new LogEmoji($data['emoji']);
70
71         $envelope = new Envelope($message);
72
73         // needed only if you need this to be sent through the non-default
   bus
74         $envelope = $envelope->with(new BusNameStamp('command.bus'));
75
76         return $envelope;
77     }
78 }
```

Cool! That created a private function down here with that code. I'll add an `array` type-hint. Back in `decode()`, we're already calling this method. So, no big change.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
↕ // ... lines 13 - 31
32     private function createLogEmojiEnvelope(array $data): Envelope
33     {
↕ // ... lines 34 - 44
45     }
46 }
```

## Using Headers for the Type

The *key* question is: if multiple *types* of messages are being added to the queue, how can the serializer determine which *type* of message this is? Well, we could add maybe a `type` key to the JSON itself. That might be fine. But, there's *another* spot on the message that can hold data: the *headers*. These work a lot like HTTP headers: they're just "extra" information you can store about the message. Whatever header we put here will make it *back* to our serializer when it's consumed.

Ok, so let's add a new header called `type` set to `emoji`. I just made that up. I'm not making this a class name... because that external system won't know or care about what PHP classes we use internally to handle this. It's just saying:

> *“This is an "emoji" type of message”*

Back in our serializer, let's first check to make sure that header is set: if not `isset($headers['type'])`, then throw a new `MessageDecodingFailedException` with:

> *“Missing "type" header”*

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 23
24         if (!isset($headers['type'])) {
25             throw new MessageDecodingFailedException('Missing "type"
   header');
26         }
↕ // ... lines 27 - 33
34     }
↕ // ... lines 35 - 54
55 }
```

Then, down here, we'll use a good, old-fashioned switch case statement on `$headers['type']`. If this is set to `emoji`, return `$this->createLogEmojiEnvelope()`.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 23
24         if (!isset($headers['type'])) {
25             throw new MessageDecodingFailedException('Missing "type"
   header');
26         }
27
28         switch ($headers['type']) {
29             case 'emoji':
30                 $envelope = $this->createLogEmojiEnvelope($data);
31                 break;
↕ // ... lines 32 - 33
34         }
↕ // ... lines 35 - 43
44         return $envelope;
45     }
↕ // ... lines 46 - 90
91 }
```

After this, you would add any other "types" that the external system publishes, like `delete_photo`. In those cases you would instantiate a *different* message object and return

that. And, if some unexpected "type" is passed, let's throw a new `MessageDecodingFailedException` with

> *“Invalid type "%s"”*

passing `$headers['type']` as the wildcard.

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
13     public function decode(array $encodedEnvelope): Envelope
14     {
↕ // ... lines 15 - 27
28         switch ($headers['type']) {
29             case 'emoji':
30                 $envelope = $this->createLogEmojiEnvelope($data);
31                 break;
32             default:
33                 throw new MessageDecodingFailedException(sprintf('Invalid
   type "%s"', $headers['type']));
34         }
↕ // ... lines 35 - 90
91 }
```

**Tip**

To support retries on failure, you also need to re-add the "type" header inside `encode()`:

src/Messenger/ExternalJsonMessageSerializer.php

```
↕ // ... lines 1 - 10
11 class ExternalJsonMessageSerializer implements SerializerInterface
12 {
↕ // ... lines 13 - 46
47     public function encode(Envelope $envelope): array
48     {
↕ // ... lines 49 - 53
54         if ($message instanceof LogEmoji) {
↕ // ... lines 55 - 56
57             $type = 'emoji';
↕ // ... lines 58 - 59
60         }
↕ // ... lines 61 - 66
67         return [
↕ // ... line 68
69             'headers' => [
70                 // store stamps as a header - to be read in decode()
71                 'stamps' => serialize($allStamps),
72                 'type' => $type,
73             ],
74         ];
75     }
↕ // ... lines 76 - 90
91 }
```

Kinda cool, right? Let's go stop our worker, then restart it so it sees our new code:

```
php bin/console messenger:consume -vv external_messages
```

Back in the Rabbit manager, I'll change the `emojis` key back to `emoji` and... publish! In the terminal... sweet! It worked! Now change the `type` header to something we don't support, like `photo`. Publish and... yea! An exception killed our worker:

> *"Invalid type "photo"."*

Ok friends... yea... that's it! Congrats on making it to the end! I hope you enjoyed the ride as much as I did! I mean, handling messages asynchronously... that's pretty fun stuff. The *great* thing about Messenger is that it works brilliantly out of the box with a *single* message bus and the Doctrine transport. Or, you can go *crazy*: create multiple transports, send things to RabbitMQ, create custom exchanges with binding keys or use your own serializer to... well... basically do *whatever* you want. The power... it's... intoxicating!

So, start writing some crazy handler code and then... handle that work later! And let us know what you're building. As always, if you have some questions, we're there for you in the comments.

Alright friends, seeya next time!

*With <3 from SymfonyCasts*